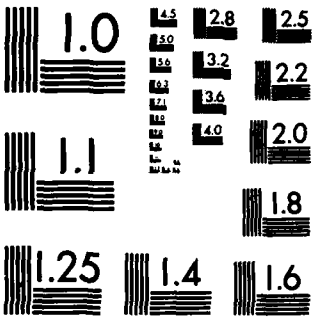
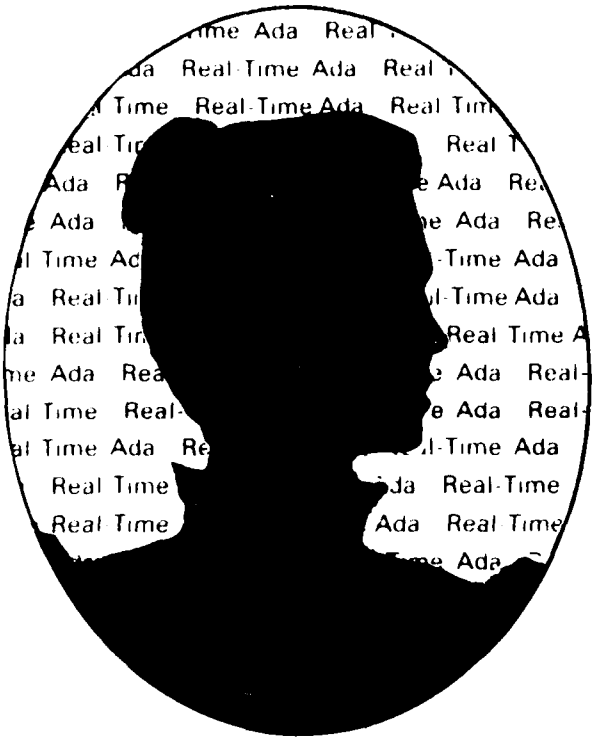
AD-A146 258 REAL-TIME ADA (TRADEMARK)(U) SOFTECH INC WALTHAM MA 1/4
JUL 84 DAAB07-83-C-K514

UNCLASSIFIED F/G 9/2 NL



JULY 1984

Real-Time Ada®



Center For Tactical Computer Systems
(CENTACS)

U.S. Army Communications- Electronics Command
(CECOM)

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

Contract DAAB07-83-C-K514

® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

PREFACE

The Real-Time Ada Workbook was developed under Contract Number DAAB07-83-C-K514 for the U.S. Army, Center for Tactical Computer Systems (CENTACS), Fort Monmouth, New Jersey.

This workbook is organized as a series of Exercises for the Reader. Each exercise appears in the following format.

First the objective of the exercise is stated. It will prepare the students for the exercise by informing them what they will learn from the exercise.

A Tutorial section follows. This section provides the necessary background information for working the problem.

The Tutorial is followed by the Problem Statement. Here the student is given a situation in which he is to assume the role of "problem solver," such as programmer, system designer, or maintainer. The problem may consist of a set of short answer questions or may require the writing or design of Ada code.

Finally, there is a Discussion and Solution section. The style of this section depends on the nature of the Problem Statement. When the Problem consists of a set of questions, then this section analyzes the answers provided. For problems that require Ada code, a brief discussion introduces the Ada solution. Portions of the code are then later discussed in depth.

This workbook parallels and can be used in conjunction with L401, Real Time Systems in Ada, of the U.S. Army Model Ada Training Curriculum. It is also intended as a follow-on for the Advanced Ada Workbook. It assumes the reader is familiar with all the concepts covered in both the Ada Primer and the Advanced Ada Workbooks.

The first chapter of this workbook presents the language independent concepts of concurrent programming. The remaining five chapters discuss the Ada features which enable the programmer to write concurrent programs.

TABLE OF CONTENTS

A1

CHAPTER 1

CONCURRENT PROGRAMMING CONCEPTS

1.1 Concurrent Programming Concepts

EXERCISE 1.1

CONCURRENT PROGRAMMING CONCEPTS

Objective

This section is designed to introduce the concepts of abstract concurrent programming, without reference to any particular language. The notion of process is explained, and a view of program execution as consisting of one or more processes is introduced. Some of the problems that have to be considered here, such as synchronization and mutual exclusion, are dealt with briefly. Concerns like deadlock and fairness are also introduced.

Tutorial

Concurrent programming models systems where several activities are proceeding in parallel. In the real world, concurrent programming helps describe many situations where several threads of control execute more or less in an interrelated fashion. In the realm of computers, concurrent programming is necessary in operating system design, and increasingly in programming distributed systems. Traditionally, concurrent programming has depended on low-level constructs like semaphores. However, many modern programming languages provide higher-level, structured constructs like Ada's tasks, Concurrent Pascal's processes and Modula's modules.

Concurrent programming is inherently more complex than simple sequential programming since the various activities may affect each other in unexpected ways. The relevant issues include asynchronism and mutual exclusion. These arise because the relative progress of different threads of control is not necessarily known a priori, and because the different activities might need to use the same resources. It is necessary to keep in mind certain pitfalls like deadlock, where different threads of control wait for the same resources, bringing the entire system to a complete halt.

Processes

The "process" concept is useful in this context. A process is defined as the abstract entity that moves through the statements of a sequential program, executing them in turn. It corresponds to the intuitive idea of a thread of control. Thus, for a program executing on a computer, the sequence of actions involved with the execution can be considered a process. A process has a state associated with each point in its execution sequence, and can be completely described by this state information. In many modern computers, multiprogramming is available, i.e. there are several programs being executed on the same processor. In this case, each non-concurrent program can be associated with a single process. In fact, most operating systems follow this concept; the only entities in the system are processes which can communicate with each other, request and release resources, and have lifetimes during which they are initiated and terminated. Concurrent programs which have several threads of control will have several processes associated with them.

The issue of how several processes can run simultaneously on a single processor is worth considering. Typical hardware can only execute one instruction at any given time. On such a machine there can be no physical concurrency, i.e. only one process can be active at any time. However, the illusion of concurrency can be provided by interleaving the executions of these processes. In other words, several processes share the processor. Each process is allocated a short time (called a time slice) on the processor, so it would appear over a period of time that the processes are indeed running concurrently. Usually the operating system provides this facility in its processor management algorithms. In the case of Ada, as we shall see later, the runtime support system provides this function. Given this implementation of logical concurrency, the rate at which a given process proceeds is not known. Therefore no assumptions can be made about the progress of the execution of a given program. Such processes are called asynchronous.

The asynchronous nature of the processes would have no impact if all the processes executed completely independently of each other and of external input. However, if it is necessary that the processes cooperate in order to perform some action, then they must be able to communicate and synchronize. The two most common methods by which inter-process communication is achieved are by using shared variables and by message passing. In the former case, as the name implies, two (or more) processes modify and examine the same global variable to communicate with each other. In the latter case, messages between processes are explicitly sent and received or left in "mailboxes" known to both processes.

Process Synchronization Using Shared Variables

As a simple example of a shared-variable scenario, consider a situation where two processes A and B each have access to the variable x, which they write into and read from. Let us say that process A prints x and increments it by 1, and that process B prints x and increments it by 2:

process A		process B	
`Put (x);`	`-- statement 1`	`Put (x);`	`-- statement 3`
`x := x + 1;`	`-- statement 2`	`x := x + 2;`	`-- statement 4`

The statements are numbered for ease of referring to them. Let us suppose that the initial value of x is 0. As we pointed out above, no guarantees can be made about the relative speeds of the two processes, so that we cannot predict the order in which these statements will be executed. In one case, the order may be 1,3,2,4 -- so that two zeros will be printed and that x will be set to three. In another case, the order may be 1,2,3,4 -- with the result that a zero and a one will be printed and x set to three. If the order is 3,4,1,2, then a zero and and a two will be printed and x set to three. Thus the result of the execution of the two processes is not consistent and reproducible, since the underlying implementation of logical concurrency becomes important.

Another point may be made about the above example, which is that we have used the concept of an "atomic action" here -- the increment statement has been assumed to be indivisible. In reality, most compilers would translate the statement

```
x := x + 1;
```

into several assembly language instructions similar to the following:

```
LOAD x (bring the value of x into the accumulator)
ADD  1  (add one to the accumulator)
STORE x (put the value in the accumulator into the location of x)
```

This brings about the possibility of further interleavings, since in processes A and B, these instructions can also overlap, with the result that x might not always get the final value of three. For instance, if the LOAD and ADD instructions for A are executed, then the LOAD and ADD for B, the STORE instruction for A, and finally the STORE instruction for B, the final value of x will be two. (Consider what would happen if the STORE instructions occurred in the opposite order.)

The above example illustrates what is called a "race condition", a situation in which the relative speed with which processes are executed influences the outcome of program execution. Race conditions can be avoided by making certain actions atomic and allowing only one such action to occur at a time. For instance, in our example above, all accesses to x must be made indivisible. A sequence of statements that must appear to be atomic and indivisible is called a "critical region" or "critical section". In our example, the code segment for process A is a critical region for A, and similarly for B. Thus when process A is executing a critical section manipulating some shared resource (such as the variable x), process B must not be executing a critical region manipulating the same resource. Thus manipulation of the resource by process A and by process B are mutually exclusive. "Mutual exclusion" means that critical regions are to be executed by processes without interference from each other.

Busy waiting is the most obvious way in which mutual exclusion can be implemented. As the name implies, a process wishing to perform operations in a critical section manipulating some shared resource will await an event which will signal the fact that all other processes have ceased accessing the resource. This event is usually the setting of a "lock variable." Therefore the process keeps checking the value of the lock variable until it gets an "unlocked" value. This is a waste of processor resources because processor time slices are taken up by a process even when it can do no useful work. Furthermore, simple busy waits do not guarantee mutual exclusion, since the test-and-set nature of the operation on the lock variable is error-prone. To guarantee mutual exclusion, it is necessary to use careful exit and entry protocols along with the lock variable. Dekker's algorithm, the details of which we shall not go into, is an example of such a protocol. The general format of such protocols can be shown thus:

Process A	Process B
loop	loop
entry protocol;	entry protocol;
critical section;	critical section;
exit protocol;	exit protocol;
noncritical section;	noncritical section;
end loop;	end loop;

The critical sections consist of all code that involves the global variable, and the noncritical sections of all the other code. Ada tasking constructs provide an equivalent construct, as will be seen in subsequent chapters.

Process Synchronization Using Semaphores

Semaphores, which were first described by E. W. Dijkstra, are a more structured synchronization primitive. Semaphores are special variables whose values are non-negative integers. Only two operations are defined on semaphores, the P and V operations (also known as the wait and signal operations respectively). For a semaphore s, the operations are defined thus:

```
P (s) : wait until s > 0; then do s := s - 1 (test and decrement
                                               indivisibly)
V (s) : s := s + 1 (increment indivisibly)
```

The P operation, when performed by the running process, may block it, that is, make the process wait; the V operation, when performed by the running process, may unblock some process that has blocked itself using the P operation previously. In any case, the process performing the V operation is not itself blocked.

In general, semaphores can have integer values; however, most of the interesting properties of semaphores can be studied using binary semaphores, which only take the two values 0 and 1. In this case, the V operation will increment s only if s is 0.

The exit and entry protocols mentioned above for busy waiting can be implemented somewhat simply using semaphores. The entry protocol (for both the processes) is merely P (Sem) where Sem is a semaphore variable global to both processes; the exit protocol is merely V (Sem). It can be demonstrated that this will provide the required mutual exclusion.

Semaphores can be implemented using busy waiting. This is justified in situations where busy waiting can be tolerated. However, semaphores are usually provided by the system kernel. The kernel is the most fundamental module in the operating system, and provides the abstraction of processes by supporting processor management. The kernel also supports device interrupt handling and interprocess communication. Processor management, or the sharing of the CPU between the various processes in the system, is usually based on several queues, such as the ready queue, the blocked queue and so on that processes are moved between. The kernel supports semaphores by attaching queues to semaphore variables as well, so that processes can be made to wait on them.

Process Synchronization Using Monitors

Monitors are another approach to the synchronization problem. In this case the onus of providing mutual exclusion is shifted away from the user to the implementer of the construct. Monitors are essentially resources and data structures on which certain predefined operations are available. We shall speak in greater detail about the Ada tasking implementation of monitors in Chapter 4. The implementation of the monitor is not made visible to the user. The operations available on them are procedure calls often called "entries." These entries are made mutually exclusive by the use of synchronizing calls, often wait and signal, to certain variables in the monitor that are restricted to be mutual exclusion devices. The wait and signal operations are quite similar to semaphore P and V operations. A monitor may have several entries; each entry may have several processes associated with it. Providing that they are not accessing shared data, more than one process can be in a monitor at a given time. The user merely makes calls to the entries in a fashion similar to procedure calls, and the underlying implementation takes care of the mutual exclusion problems, blocking and unblocking processes as need be. The programming languages Concurrent Pascal and Modula-2 provide monitors as their main concurrency constructs.

Process Synchronization Using Message Passing

A different approach to interprocess communication makes use of message passing instead of shared variables. Message passing effects transfer of data in addition to providing mutual exclusion and synchronization. Processes use SEND and RECEIVE commands to communicate with each other. In general the SEND command will specify both the name of the receiver and the information to be sent; symmetrically, the RECEIVE command will specify the name of the sender and variables to receive the information. The parallels between these commands and ordinary procedure calls are many; in fact, under some conditions, message passing can be viewed as remote procedure calls. The most general form of the commands then will be:

```
SEND <information> to <receiver>;     and
RECEIVE <variables> from <sender>;
```

One of the issues to be resolved in message passing is whether the sender or the receiver process is blocked awaiting the completion of the operation. In other words, how does synchronization take place between the sender and the receiver? One possibility is to perform synchronous message passing, in which case a sender will be blocked until some other process is willing to receive its message (receiver will be similarly blocked as well). This is the alternative favored by Ada's rendezvous mechanism. Another option would be asynchronous communication, where a limited buffering is offered, so that a sender can send several messages before the receiver has absorbed the current message.

Another design issue is the naming of the processes. If, as indicated above, both the receiver and sender are named directly, then the message passing channel is to be used strictly for communicating between the two named processes. This is simple and desirable in many cases, but there are certain paradigms in multi-process systems which are difficult to implement using this scheme. For instance, it is sometimes necessary to broadcast the same information to many or all processes in the system, regardless of the identity of the process. If direct naming is used, a separate channel will have to be created for each transaction. Another common paradigm is the server/user scenario. In this case, some process provides a service to whoever needs the service. For instance, a terminal handler process would be such a server, and any user process that desired to send information to the terminal would have to go through it. Thus the server should provide a non-specific service, i.e. it should be a receiver that does not name the sender. It should have a receive statement of the form:

```
RECEIVE <information>;
```

Thus there are some instances where direct naming is not a suitable mechanism. Furthermore, we have been considering symmetric mechanisms until now. There is no reason why perhaps the SEND shouldn't be specific (i.e. it names the receiver) while the RECEIVE is non-specific. In fact this is the Ada view of the rendezvous.

Another issue with message passing is the question of where the messages are to be deposited. A common paradigm involves "mailboxes" which are global variables updated by processes to provide asynchronous communication. These are especially suitable for such situations as the producer/consumer scenario in which a producer process produces some output which is consumed by a consumer process. The mailbox implementation of this involves a global mailbox visible to both these processes, and a send operation by the producer into this mailbox. The consumer then performs a receive operation on the mailbox to retrieve the data. One of the problems that mailbox implementations have is the size of the mailbox. If fixed-size mailboxes are used, all messages will have to be padded or reduced to that size. More general, arbitrary-sized mailboxes are more complicated. In addition, if the mailboxes are created dynamically at runtime, it is possible to provide some elegant facilities for communication. One such is the "pipeline" paradigm, wherein a series of processes pass their outputs to others in a fashion quite similar to a pipeline. Unix's "|" operator, which provides piping, is a particularly useful example of this.

Now that we have considered the general concepts of concurrent programming, and discussed several implementation details, it is necessary to touch upon some of the pitfalls of concurrency. Foremost among this is the notion of deadlock.

Deadlock: Avoidance and Detection

"Deadlock" is a situation that arises if a process awaits an event that can never happen. The result is, at best, wastage of CPU and memory resources, and at worst, a complete halt to the system. It is a serious problem, especially in critical real-time systems such as process control and mission control applications, where the loss of the system can have catastrophic consequences. There are many ways in which deadlock can happen, and usually there needs to be external interference in the form of operator intervention or system shutdown to get the system out of deadlock. As far as is known, no existing programming language provides guarantees that programs written in it will be deadlock-free. It is up to the applications programmer to structure the software such that deadlock is unlikely. There are several deadlock prevention and deadlock detection algorithms which we shall touch upon briefly.

Perhaps the simplest example of deadlock is the classic two-process/two-resource problem. Consider a situation where there are two processes P1 and P2 and two resources R1 and R2. Suppose now that P1 has R1 allocated to it, and P2 has R2. Furthermore, say P1 has reached a position where it needs R2 as well to continue. Therefore it makes a request for R2 and blocks itself. However, P2, which has control of R2, is not ready to relinquish it until it performs some action requiring R1. Thus each process covets a resource owned by the other, and therefore they are both blocked. This model assumes of course that allocated resources cannot be forcibly retrieved unless the "owning" process relinquishes them voluntarily. At this point, processes P1 and P2 are deadlocked and unable to continue. This example might explain why an earlier term for deadlock was "deadly embrace."

When considering resources, it is important to make a distinction between reusable and consumable resources. The resources that can cause deadlock are usually reusable resources such as the printer, memory etc. Consumable resources like messages usually present no problem. According to Coffman et al, it can be shown that deadlock on reusable resources arises only if certain conditions are satisfied: mutual exclusion, non-preemptive resource allocation, partial resource allocation, and circular waiting. Mutual exclusion in this context means that resources can only be used by one process at a given time. Non-preemptive resource allocation means that, as described above, a resource can only be released by the process that acquired it. Partial allocation means that the resources needed by a process are not necessarily allocated all at once, but may be allocated one by one. Circular waiting means that processes can acquire some of their resources and be blocked awaiting other resources. To avoid deadlock, it will be sufficient not to satisfy one of these conditions. Non-mutually-exclusive access to resources is difficult to provide, considering that some are readable and writeable objects. Preemptive allocation of resources is also not always practical, since a resource might be in an inconsistent state when the process that acquired it is pre-empted. Complete allocation of all resources in advance is obviously inefficient. It is practicable, however, to avoid circular waiting by using a constrained allocation scheme, often called the "Banker's algorithm", which essentially takes a rather pessimistic view of things. We shall not go into the details, but the Banker's algorithm works as follows: the maximum requirements of any process is known ahead of time; the algorithm doles out resources such that at no time is there the possibility of deadlock. It deduces this by considering the effect of each allocation on the

potential for deadlock. In the worst case, this might dictate a purely sequential execution of the concurrent processes. In addition, the algorithm is rather expensive.

Another deadlock-avoidance algorithm depends on a hierarchical structuring of the resources. There are several levels in the hierarchy, and a process is constrained to request resources from a particular level all at once. Furthermore, a process that has acquired resources at a level j can only request further resources at a higher level k, $k > j$. On releasing the resources, the opposite order is followed -- resources at level k, $k > j$, must be released before those at level j. It can be shown that under these conditions, deadlock can be avoided.

It is also possible to detect deadlocks. One deadlock-detection algorithm, due to [Holt], considers a graph which summarizes the processes, resources and requests in the system. By deleting a process from the graph if its resource requests can be granted, it can be deduced whether or not deadlock is possible. If all processes can be deleted from the graph, then it means that the system is deadlock-free. It is possible to use this algorithm to detect suspected deadlocks, and it is also possible to prevent potential deadlocks by applying this algorithm at each resource allocation.

Fairness

Fairness is another issue in concurrent programming. This usually applies to the scheduling policy, which is said to be fair if there is no situation in which some process is repeatedly not chosen. In other words, every process gets an occasional CPU time slice, and every process can thus make finite progress. Indefinite overtaking on the other hand, can happen in certain situations where scheduling is priority-driven. Process starvation, as this is sometimes termed, can be explained picturesquely in terms of the famous Dining Philosophers Problem, where a philosopher who is continually denied access to the two forks (or chopsticks) with which to eat will literally starve. A simple FIFO (First in, first out) queue is fair, since no process will be stuck for ever.

Though unfairness might not appear to be as serious a problem as deadlock, its effects on the unfortunate process that is indefinitely overtaken are just as disastrous. Thus it behooves the system programmer to ascertain that neither of these pathological situations arises in the system under construction.

This concludes our introductory section on the perils and joys of concurrent programming.

Exercises

1. Consider the following simple implementation of binary semaphores using busy waiting. Is this a satisfactory implementation? If so, explain how. If not, explain why not.

```
package Binary_Semaphore_Package is

    type Binary_Semaphore_Type is access Boolean;

    procedure P (Go_Ahead : in out Binary_Semaphore_Type);

    procedure V (Go_Ahead : in out Binary_Semaphore_Type);

end Binary_Semaphore_Package;


package body Binary_Semaphore_Package is


    procedure P (Go_Ahead : in out Binary_Semaphore_Type) is
    begin

        while (Go_Ahead.all = False) loop
              null;  -- busy wait
        end loop;
        Go_Ahead.all := False;

    end P;

    procedure V (Go_Ahead : in out Binary_Semaphore_Type) is
    begin
        -- Go_Ahead is in out to allow assignment to the allocated
                                -- object.  If it were an out
                                -- parameter, only an access value
                                -- could be assigned to Go_Ahead.
        Go_Ahead.all := True;

    end V;

end Binary_Semaphore_Package;
```

2. Assume that a concurrent programming language provides you with the following operations on monitor variables: wait and signal. The monitor Monitor_Var is visible only within the body of the monitor itself. The wait operation works as follows: the invoking process will be suspended. The signal operation works as follows: if no process is suspended on the Monitor_Var, then the signaller continues; otherwise, one of the suspended processes is woken up. Thereafter either the signaller or the newly awakened process continues. This is not necessarily the semantics of wait and signal in regular monitor implementations: we use it here for pedagogical purposes. Consider the following program fragment which purports to implement a single-element buffer. If you think the program is correct, explain how it works; otherwise, describe what is wrong with it.

```
with Element_Decl_Package;      -- Type Element_Type is declared
use Element_Decl_Package;       -- in this package
package Monitor_Package is

        procedure Put (Info : in Element_Type);

        procedure Get (Info : out Element_Type);

end Monitor_Package;

with Monitor;                   -- generic package implementing
                                -- monitors.  It has the
                                -- predefined wait and signal
                                -- operations
package body Monitor_Package is

        package Monitor_Var is new Monitor;


        Element : Element_Type;

        procedure Put (Info : in Element_Type) is
```

```
        begin

            Monitor_Var.wait;
            Element := Info;
            Monitor_Var.signal;

        end Put;

        procedure Get (Info : out Element_Type) is

        begin

            Monitor_Var.wait;
            Info := Element;
            Monitor_Var.signal;

        end Get;

    end Monitor_Package;
```

3. No discussion of concurrency can escape the Dining Philosophers Problem. The problem centers on five philosophers who spend their days alternately eating and thinking. They are seated at a table with a bowl of spaghetti in the center of the table and a plate in front of each of them. Between every two philosophers is a chopstick. Each philosopher needs two chopsticks at any time to pick up some food from the bowl. However, philosophers may only pick up chopsticks adjacent to themselves. Each philosopher returns the chopsticks to the table after eating. The problem is to ensure a) that deadlock does not occur, i.e. every philosopher does not pick up one chopstick and b) the system is fair, i.e. no philosopher is starved because of greedy neighbors.

It is possible to come up with a deadlock-free solution using semaphores by insisting that a philosopher pick up any chopsticks only if two are available. However, this does nothing to prevent starvation of some unfortunate philosopher. Consider the following scenario. Philosopher P1 and P3 pick up two chopsticks each and are using them. P2 now comes along, finds that he doesn't have two chopsticks available, and therefore proceeds to think. In the meantime, P1 lays her chopsticks down, thinks for a while, and then picks them up again. Then, P3 lays his chopsticks down, thinks and picks them up again. During this entire sequence, P2 has not had two chopsticks available, and has been unable to eat. Now this sequence can continue indefinitely, and poor P2 will starve, being overtaken all the time. Can you suggest some methods by which fairness can be introduced into the system?

Solutions and Discussion

1. Though this appears to be satisfactory, in that the process invoking the P procedure will be delayed until Go_Ahead is finally True, there are some problems due to potential concurrent access. For instance, consider the possibility that there are several processes all of whom have executed P operations, and therefore continually poll Go_Ahead to see if it is True. Suppose such a process, P1, first observes the fact that Go_Ahead is True, and therefore proceeds to set Go_Ahead to False in preparation for continuing. However, it is possible that before P1 sets Go_Ahead to False, process P2 observes Go_Ahead, sees that it is now True, and therefore also decides to continue. This would then mean that P1 and P2 both enter their critical regions under the assumptions that they are the only processes modifying the shared data. Thus the semaphore has not achieved mutual exclusion.

2. Under two conditions, our monitor will work: if Monitor_Var has been initialized to be open, and the wait and signal primitives are indivisible. The Put operation will wait till Monitor_Var has been signalled, and when it is finished, it will signal Monitor_Var; similarly the Get operation will wait as well. If two Put operations are attempted in a row, the second will not be permitted until there has been a Get operation, and vice versa.

3. The simplest way of introducing fairness would be to attach priorities to the philosophers: the higher the number of meals they have compared to the average number of meals, the lower the priority of their next request. Further, some FIFO behaviour can be simulated: earlier requests will gain priority over later requests.

Bibliography:

Andrews, G. et al., Concepts and Notations for Concurrent Programming, pp. 3-44, Computing Surveys, March 1983.

Brinch Hansen, P., Operating System Principles, Prentice-Hall, 1973.

Coffman, System deadlocks, pp. 67-78, Computing Surveys, June 1971.

Holt, R.C., Some Deadlock Properties of Computer Systems, pp, 178-196, Computing Surveys, September 1973.

Holt, R.C. et al., Structured Concurrent Programming with Operating System Applications, Addison-Wesley, 1978.

CHAPTER 2

ADA TASKING CONCEPTS

2.1 Ada Tasking Concepts

EXERCISE 2.1

ADA TASKING CONCEPTS

Objective

This exercise is designed to explain the basic tasking concepts, in particular the notion that Ada tasks are objects. This tutorial covers task declarations, and the activation and termination of tasks.

Tutorial

In order to write a program consisting of concurrent processes, the software engineer needs some way of specifying which processes are conceptually concurrent. In Ada, the language construct used to group such sequences of actions is known as a task. Because a task is defined in terms of actions rather than statements or instructions, even the execution of a simple program, such as a procedure which prints "Hello" on a terminal, can be viewed as a single, implicit task whose thread of execution runs in parallel with the rest of the system (i.e. operating system, printer, disk drives, peripherals, etc.). In other words, running this greeting procedure is conceptually equivalent to having a task which prints this message.

Tasks can run simultaneously on different processors, as in a distributed system, or they can be interleaved, as in a single-processor

time-sharing system. Declaring a program unit to be a task provides only logical concurrency; the language makes no guarantees about the speed of various tasks. The operating or runtime system has the responsibility for scheduling different tasks and for allocating any resources which they may need.

Tasks are independent program units: they run at their own pace and are essentially isolated, unless the programmer specifies explicit synchronization points. A synchronization point serves as a place where one of the affected tasks can "wait" for the other process to "catch up" to it, i.e. arrive at the corresponding point in its sequence of actions. The synchronization mechanism delimits that section of code which is to be executed while the tasks are synchronized. The programmer can, for example, use this opportunity to exchange data between two tasks. Note that any given synchronization point involves exactly two processes. The synchronization mechanism is known as the rendezvous and is discussed in detail in Exercise 3.1.

In Ada, tasks allow the programmer to decompose a problem into several independent threads of control. This technique enables him to model different activities in the real world simultaneously. For example, an avionics system has altitude, radar, and velocity sensors, a gyroscope, and a graphics display, each of which are continuously monitored for valid readings. Additionally, the graphics display is updated periodically to reflect changes in position, altitude, velocity,

and terrain. Each of these subsystems can be modeled by a task. These tasks are independent activities: the computations used to update the graphics display in the cockpit run without help or resources from the sensor tasks. This independence does not preclude the exchange of information between these tasks -- the programmer must provide some way for the new altitude readings to be sent to the display task.

To ensure reliability, this avionics system has redundant sensors and displays, each of which executes an identical sequence of actions. In other words, for each type of sensor, there are several objects of that type, namely the actual sensors. They share the same operational instruction sequence; however, they each have a private data area in which to manipulate and record actual readings.

The sensors described above may be specified in Ada by a task type declaration and corresponding task body. Consider the specific case of the altimeter. The class of sensors is:

```
task type Altitude_Sensor_Type;
```

Four tasks, one for each sensor, can now be created from the blueprint of the task type Altitude_Sensor_Type:

```
Main_Altitude_Sensor : Altitude_Sensor_Type;
Backup_Altitude_Sensor_1,
     Backup_Altitude_Sensor_2,
     Backup_Altitude_Sensor_3 : Altitude_Sensor_Type;
```

Once these task objects are activated, they execute concurrently. Each of these tasks behaves in an identical manner, as specified by the code in the task body, shown below. Another process can compare the altitude readings of these four sensors and determine possible sensor failure.

In defining the task body for the sensors, certain assumptions regarding the hardware configuration are made. Low level I/O (at the device level) is used to obtain the latest sensor reading. Each sensor is "tied" to a particular device, and a device assignment procedure is provided accordingly. The five most recent sensor readings are averaged, and they are output to the appropriate device, corresponding to some port used by the sensor task and another part of the avionics system. First, certain types, objects and procedures are declared in the package Altimeter_Definitions.

```
with Low_Level_IO;
package Altimeter_Definitions is

     type Altitude_Reading_Type is range 0 .. 500_000;
     subtype Valid_Altitude_Type is Altitude_Reading_Type
                    range 0 .. 100_000;
     procedure Get_Device_Assignment
                    (Device : out Low_Level_IO.Device_Type);
     procedure Get_Sensor_Reading
                    (Device  : in Low_Level_IO.Device_Type;
                     Reading : out Valid_Altitude_Type);
     procedure Put_Average_Reading
                    (Device  : in Low_Level_IO.Device_Type;
                     Reading : in Valid_Altitude_Type);

end Altimeter_Definitions;
```

The package body for Altimeter_Definitions is not given.

```
with Altimeter_Definitions; use Altimeter_Definitions;
with Low_Level_IO; use Low_Level_IO;
separate (Operate_Avionics)  -- procedure to be introduced
                             --  later in Tutorial
task body Altitude_Sensor_Type is

     New_Reading     : Valid_Altitude_Type;
     Device          : Low_Level_IO.Sensor_Type;
     Average_Reading : Altitude_Reading_Type;
     Altitudes : array (1 .. 5) of Valid_Altitude_Type :=
                    (0, 0, 0, 0, 0);

begin -- Altitude_Sensor_Type task body

     Get_Device_Assignment (Device);
     loop  -- forever
          Get_Sensor_Reading (Device, New_Reading);
          Average_Reading := 0;
          Altitudes := New_Reading & Altitudes (1 .. 4);
                    -- replace oldest reading with new one
          for I in Altitudes'Range loop
               Average_Reading :=
                    Average_Reading + Altitudes(I);
          end loop;
          Average_Reading := Average_Reading /
                  Altitude_Reading_Type (Altitudes'Length);
          Put_Average_Reading (Device, Average_Reading);

     end loop;

end Altitude_Sensor_Type;
```

All tasks in Ada are treated as objects of some task type. In general, this type is a named type declared by a task type declaration:

```
task type Task_Type_Name [is
     {entry Entry_Name;}
end Task_Type_Name];
```

Within the type declaration only a single kind of construct is allowed, known as an entry declaration. Entry declarations name the points within the task at which another task may communicate with the task. Entries are discussed in detail in Exercise 3.1.

Insofar as tasks are objects, they follow the Ada rules for objects, i.e. they are declared to be of some previously declared task type. Furthermore, because tasks belong to a task type, they may be used in places where objects are allowed, such as in a component declaration of an array or record type, or as the designated object of an access type. For instance, if a task type is used to represent a tank maneuvering on the field, then rather than declare 50 tank task objects to model the 50 tanks deployed, a 50-element array can be declared whose component is the tank task type:

```
task type Tank_Type is
     entry Com_Channel_1 (Data : in Data_Type);
     entry Com_Channel_2 (Message : out Message_Type);
     entry Fire;
     entry Danger;
end Tank_Type;
type Battalion_Type is array (1 .. 50) of Tank_Type;

Tank_Battalion : Battalion_Type;

task body Tank_Type is
     ...   -- local data
begin  -- Tank_Type

     ...   -- statements defining a sequence of actions

end Tank_Type;
```

Now, each element of the array Tank_Battalion is itself a task object executing the sequence of actions defined inside the task body of Tank_Type. Furthermore, any other part of the system may communicate to any individual tank task through the four synchronization points declared by the entries. Note that the entry Com_Channel_1 of Tank_Battalion(I) is not the same entry as Com_Channel_1 of Tank_Battalion(J) (assuming I different from J), so that a communication on channel 1 to the ith tank is only received by this ith tank.

Task types are limited types, and therefore it is illegal to assign them, compare them for equality, and so forth. If they are used as parameters in subprograms, then they may not be out mode parameters. The only other allowed operations on tasks involve calling an entry in a task object, a topic discussed in Exercise 3.1. Given that task types are limited types, any composite type one of whose components is a task type is itself a limited type, and all the rules described in this paragraph apply. The previous example, Battalion_Type, is a limited type.

In the same way that it is possible to declare an array object without declaring it to be of an explicit type, one can declare a task object without naming its type. Such a task is said to have an anonymous task type. Its declaration is similar to that of a task type, containing any necessary entry declarations but omitting the reserved word type:

```
task Task_Name [is
    {entry Entry_Name;
     entry Entry_Name;}
end Task_Name];
```

For example, suppose that a system has a single clock which prints out the time at 300 second intervals on the system console, as a way of notifying the operator that the system is still operational. The corresponding task declaration is:

```
task Clock;
```

The body of this task contains the sequence of actions to be executed while this task is running.

```
with Text_IO; use Text_IO;

task body Clock is

     type Time_Type is range 0 .. 86400;
          -- range covers number of seconds per day
     System_Time : Time_Type;
     pragma Shared (System_Time);
     for System_Time use at 16#000F#;
          -- the hardware will put values into this
          --  variable at the correct location

begin   -- Clock

     loop
          Put_Line (Time_Type'Image (System_Time));
          delay 1.0;
     end loop;

end Clock;
```

Notice how the variable System_Time is declared to be a shared variable; this declaration will ensure that an optimizing compiler does generate the code necessary to read this variable at the appropriate intervals (there are no explicit assignments to this object, so a compiler could assume that as its value is seemingly never updated, its memory location need only be read the first time). The statement that reads "delay 1.0;" instructs the program to wait one minute before continuing with the next statement. This statement is discussed in more detail in Exercise 3.1. One further note on the task body of Clock: Text_IO rather than an instantiation of Integer_IO was used to output the time in order to compress the I/O into a single procedure call to Put_Line. An implicit assumption is made that the call to Put_Line is indivisible, that is that while this task is executing Put_Line, another task is not simultaneously doing another "Put" operation.

Associated with every task declaration is a task body. In other words, a body must be written for every task type, be it an explicit task type or an anonymous one. As discussed in the tank battalion example, all task objects of a given type execute the same sequence of actions, defined in the task body of that task type.

Tasks share properties both with types and with program units. The concept of task types has been illustrated in the earlier part of this tutorial. Tasks are also program units, evidenced by their structural similarity with other program units: they have two parts, a specification and a body. The specification declares interface information, in the case of tasks, the points at which a task may be "called," i.e. synchronized with some other process. The body contains executable code which is to be executed while the task is running. A task body has a declarative region as well as an executable region, and any declarations and statements normally allowed in these regions are allowed: type, object, package, task, or exception declarations; procedure calls, entry calls (to be discussed in Exercise 3.1), blocks, loops, if's, case statements, exception handlers and so on. Additionally, and unique to task bodies, accept statements may be used allowing the task to accept a call to one of its entries (more on accepts in Exercise 3.1). Unlike other program units such as subprograms, packages, and generic units, tasks may not be separately compiled. Thus, tasks may not be library units, and they may not be named in a context clause. Both task declaration and body must be placed in a declarative region. The task body may be stubbed out and later written as a subunit:

```
package Battlefield is
     .
     .
     task type Tank_Type is
          entry Com_Channel_1 (Data : in Data_Type);
          entry Com_Channel_2 (Message : out Message_Type);
          entry Fire;
          entry Danger;
     end Tank_Type;

     type Battalion_Type is array (1 .. 50) of Tank_Type;

     Tank_Battalion : Battalion_Type;
     .
     .
end Battlefield;

package body Battlefield is
     .
     .
     task body Tank_Type is separate;
     .
     .
end Battlefield;

separate (Battlefield)
task body Tank_Type is
     ...       -- local type / object declarations

begin
     ...       -- sequence of statements

end Tank_Type;
```

Up to this point, attention has focused on task structure rather than task execution. The remainder of the tutorial will address some aspects of task execution, namely activation and termination of tasks.

Task activation is the process by which tasks are put in execution. This process consists of two steps: the declarations in the task body are elaborated, then the task is free to execute the sequence of statements in its body. When there are several tasks in a declarative region, then all must be elaborated and be in a state ready to execute before execution proceeds past the "begin" of the declaring frame. Ada semantics distinguish two cases: tasks which are declared objects and tasks which are allocated objects. The rule for declared task objects states that the task becomes active at the "begin" (in the sense of the reserved word) of the frame in which it is declared. Let us look at the implications of this rule more closely, for tasks declared in procedures as well as in packages. Consider the following code segment:

```
with Text_IO; use Text_IO;
procedure Calculate_Fibonacci_Sequence is

     type Fibonacci_Type is range 0 .. 10E15;

     function Fibonacci (N : Positive) return Fibonacci_Type is
     begin
          if N < 3 then
               return Fibonacci_Type (N);
          else
               return Fibonacci (N - 1) + Fibonacci (N - 2);
          end if;
     end Fibonacci;

     task Clock;
     task body Clock is
          type Time_Type is range 0 .. 86_400;
               -- range covers number of seconds per day
          System_Time : Time_Type;
          pragma Shared (System_Time);
          for System_Time use at 16#000F#;
               -- the hardware will put values every second
               --  into this variable at this location
```

```
        begin   -- Clock

             loop
                  Put_Line (Time_Type'Image (System_Time));
                  delay 1.0;
             end loop;

        end Clock;

   begin   -- Calculate_Fibonacci_Sequence

        for F in 1 .. 100 loop
             Put_Line ("Fibonacci of " &
                       Fibonacci_Type'Image (F) & ":" &
                       Fibonacci_Type'Image (Fibonacci(F)));
        end loop;

   end Calculate_Fibonacci_Sequence;
```

The Clock task begins printing out the current time in seconds when the elaboration of the subprogram Calculate_Fibonacci_Sequence reaches the "begin" for the procedure. Then, while the subprogram is busily churning out the Fibonacci sequence, the clock task keeps time. By examining the printout, a reader could determine how fast the computer was able to calculate the sequence of Fibonacci numbers.

Now consider the case of a task declared inside a package. Refer to the Battlefield example described earlier. The "begin" of the frame in which the array of tank tasks is declared is the "begin" of the package body of Battlefield. If no "begin" is present then the task becomes active after the package body is elaborated. In the following situation then,

```
with Battlefield; use Battlefield;
procedure Operate_Command_Post is
     ...
begin
     ...
end Operate_Command_Post;
```

the task objects in the array Tank_Battalion begin running after the package body of Battlefield is elaborated. Thus the tank tasks are already active before the "begin" of Operate_Command_Post. If within Operate_Command_Post another tank task is declared,

```
with Battlefield; use Battlefield;
procedure Operate_Command_Post is
     ...
     Experimental_Tank : Tank_Type;
begin
     ...
end Operate_Command_Post;
```

then Experimental_Tank is activated at the "begin" of Operate_Command_Post, somewhat later than the activation of Tank_Battalion.

Recall that a task type may be used anywhere a type is appropriate, as in the type pointed to by an access type. Under these circumstances, the task becomes active when it is allocated. Suppose we declare the following access to the altitude sensor task and the corresponding access objects:

```
type Sensor_Type is access Altitude_Sensor_Type;
Main_Altitude_Sensor,
     Backup_Altitude_Sensor_1,
     Backup_Altitude_Sensor_2,
     Backup_Altitude_Sensor_3 : Sensor_Type;
```

Nothing happens when the "begin" for this region is reached. The programmer has an extremely fine control of the activation of each altitude sensor task. Until the allocation statement is executed, no task object starts running. Depending on the application, the task might be allocated much deeper in the program structure, say several procedure calls nested in the execution. In this particular avionics setting, turning on the altitude sensors could be the last part of the pre-take-off check performed by the pilot:

```
with Altimeter_Definitions; use Altimeter_Definitions;
procedure Operate_Avionics is

     task type Altitude_Sensor_Type;
     type Sensor_Type is access Altitude_Sensor_Type;

     Main_Altitude_Sensor,
          Backup_Altitude_Sensor_1,
          Backup_Altitude_Sensor_2,
          Backup_Altitude_Sensor_3 : Sensor_Type;

     procedure Pre_Flight_Check is separate;
     procedure Pre_Take_Off_Check is separate;
     task body Altitude_Sensor_Type is separate;

begin   -- Operate_Avionics

     Pre_Flight_Check;
     Pre_Take_Off_Check;
     -- now the altitude sensors are all active
     ...
end Operate_Avionics;

separate (Operate_Avionics)
procedure Pre_Take_Off_Check is
     ...
begin
     ...  -- check power, magnetos, wing flaps, compass,
          --  heading indicators, etc.
     -- activate altitude sensor tasks
     Main_Altitude_Sensor := new Altitude_Sensor_Type;
     Backup_Altitude_Sensor_1 := new Altitude_Sensor_Type;
     Backup_Altitude_Sensor_2 := new Altitude_Sensor_Type;
     Backup_Altitude_Sensor_3 := new Altitude_Sensor_Type;
```

```
exception   -- handle any failures from pre take-off checks
    ...
end Pre_Take_Off_Check;
```

The last issue to be addressed in this tutorial is that of task termination. Task termination refers to the actual demise of the task as opposed to the completion of its execution. A task may have executed all of the statements in its body and be sitting at the "end Task_Name;" line, in which case it is said to have completed its execution. It does not mean, however, that the task has terminated. Such a task, in fact, could be very much alive. There is a variant to the concept of completion of execution related to the terminate alternative, the details of which are presented in Exercise 3.1. Suffice it to say that when a task is waiting at a terminate alternative, it has to all intents and purposes completed its execution.

Task termination depends on the context in which the task was activated: each task depends on at least one master, which is itself some currently executing program unit or block statement. Thus a master is a procedure, function, task body, or library package. The master of a task declared in an inner package is the active frame, i.e. master, who declared this inner package. In the earlier Fibonacci example, the master of the Clock task is the subprogram Calculate_Fibonacci_Sequence. When no allocator was involved in the creation of the task object, then the master is the program unit or block

during whose execution the task object was created. To take another example, consider the Battlefield package. The tank tasks were created in the elaboration (of the body) of a library unit, package Battlefield. Thus the Tank_Battalion array of task objects depends on the package Battlefield. By contrast, the Experimental_Tank task object was created by the execution of Operate_Command_Post, so it is this procedure that is its master. In the Operate_Avionics example, the master of all four altitude sensor tasks is the Operate_Avionics procedure, not the Pre_Take_Off_Check procedure. When a task object is created through the evaluation of an allocator, then its master is the program unit (or block) which elaborated the corresponding access type definition. The access type Sensor_Type is declared in the declarative region of Operate_Avionics.

Task termination occurs when certain conditions are satisfied. The general case is addressed, so the discussion is in terms of the master as opposed to the specific instance of a task being a master. If a master has no dependent tasks and it has completed execution, then it is terminated. If a master has dependent tasks, then its termination occurs when both its execution is completed and all of its dependent tasks have terminated. Thus the Ada semantics do not allow the environment (e.g. its data) of a dependent task to disappear until it is no longer needed.

Let us reexamine the examples used in this tutorial in the context of task termination. In the Fibonacci example, the task body of Clock consists of an infinite loop; therefore as it is currently written, the

task body will never complete execution. This situation poses a dilemma for its master, the procedure Calculate_Fibonaci_Sequence. The execution of the loop statement will finish at one point, and the procedure should finish executing. If it did exit, however, then it would leave a dependent task executing, violating the semantics of the language. Thus the clock task will print out the time forever and the procedure is unable to terminate.

The Battlefield example raises some interesting issues. A library package is not a unit that can be terminated in the usual sense. However, tasks declared in a library package, such as the Tank_Battalion array, can terminate if all of their dependent tasks have terminated and they have reached the end of their sequence of statements. The main program can also terminate without waiting for Tank_Battalion to finish its mission, provided that the Experimental_Tank task, which depends on the Operate_Command_Post procedure, has terminated.

The use of allocators to determine precisely the activation of a task makes the activation independent of the environment on which the task is dependent. Thus the dependency and the activation parts of the task are decoupled. Note that the use of allocators to activate a task does not imply that removing this pointer causes the task's termination. Even should the pointer to this task object be reassigned, as in

```
Backup_Altitude_Sensor_3 := null;
```

the task continues execution. As a consequence of the above statement, the program, while remaining the master, loses all ability to communicate with this task. The program is unable to terminate until this dependent task also terminates. In the case of the altitude sensors, as the task body contains an infinite loop, it precludes its termination.

In the avionics example, the altitude sensors depend not on the procedure Pre_Take_Off_Check, their point of activation, but on the procedure Operate_Avionics, their master. Review of the task body for the task type Altitude_Sensor_Type shows that the body consists of an infinite loop. Once again, these tasks are unable to complete execution and therefore the procedure Operate_Avionics is also unable to terminate.

In order to provide for graceful shutdown of a system that uses tasks, the programmer must consider how to terminate these tasks. In some of the examples used in this tutorial, if some task communication mechanism had been available, then the master could "order" the task to complete execution. Both master and task could then have terminated. As more tasking constructs are discussed in subsequent chapters, the reader will learn to solve this problem.

Problem

One of the subsystems of a message switch physically transmits messages to their destinations. Every message sent must be logged, and the logging activity should not interfere with or delay the rest of the processing. The logging function can essentially run independently of the transmitting function, but it must have an interface with it in order to receive the necessary statistics on the message sent (i.e. the number of blocks sent, the line on which it was sent, the text of the message and its sequence number). Compare and contrast the two approaches presented below.

Approach 1

```
procedure Log (Message : in Message_Type) is
     ...
end Log;

...

loop
     Get_Next (Message);    -- Message contains all of the statistics
     Transmit (Message);    -- discussed above
     Log (Message);
end loop;
```

Approach 2

```
task type Logger is
     entry Log (Message : in Message_Type);
end Logger;

type Log_Pointer is access Logger;
Log_Access : Log_Pointer;

task body Logger is
     ...
end Logger;

...

loop
     Get_Next (Message);
     Transmit (Message);
     Log_Access := new Logger;
     Log_Access.Log (Message);  -- synchronization call with Logger
                                --  task
end loop;
```

Solution and Discussion

The first approach takes a fairly simplistic view of things, calling the procedure Log in order to log the message just transmitted. This approach, however, delays the main transmission process inasmuch as a new message can be neither read nor transmitted until the previous message is fully logged. The independence and potential concurrency of the logging procedure is not exploited.

The second solution creates a new task for each logging request. Thus no message operation needs to wait until a logger is ready to accept its information. A task terminates if it has no statements left to execute and it has no dependent tasks. All of these logging tasks will therefore die when they complete their function. Because many tasks, as opposed to a single task, are needed, a logging task type is defined. Furthermore, because these tasks should only be activated on an as needed basis, an access type to these tasks is defined. A logging task is therefore activated by the execution of an allocator. After the synchronization call, in which the logging task receives the message, no further interaction with this task is necessary, and therefore it is safe to reassign the pointer to a new logging task during the next iteration of the loop.

Creating tasks "on the fly" eliminates all unnecessary dependence between the logging funtion and the message operations. By limiting the dependence to the transfer of message information, a point which will be clearer after reading Exercise 3.1, the potential for concurrent and independent execution is best exploited. However, creating and terminating tasks may be expensive in some Ada implementations. Another solution to this problem uses buffering techniques, which will be covered in depth in Exercise 4.1.

CHAPTER 3

TASK COOPERATION

3.1 Multiple Threads of Control

3.2 Deadlock

EXERCISE 3.1

MULTIPLE THREADS OF CONTROL

Objective

In this section, ideas are developed about how concurrent programs can be written using Ada's tasking constructs. In particular, the principles of task communication and task co-operation are introduced.

Tutorial

We have seen how to declare independent tasks in section 2.1. However, tasks do, in general have to interact with one another. In section 1.1, the ideas of inter-process communication were introduced briefly. Ada uses message-passing semantics in its inter-task communication protocols. Tasks communicate by sending each other not only synchronization information, but data as well. The message-passing concept in Ada is called the "rendezvous", and to some extent conforms to the intuitive idea of two tasks that execute their code independently of each other except for the time that they "meet" and exchange information. After the rendezvous is complete, the two tasks continue independently.

The rendezvous is an asymmetric, blocking message-passing scheme. The asymmetry means that only one of the communicating tasks names the other; in particular, the "sending" task names the "receiving" task. Thus the sender specifically wishes to communicate with the receiver, but

the receiver is willing to communicate with any task. As mentioned in section 1.1, this scheme is well-suited to the server/user paradigm. The fact that this is a blocking transaction means that whichever task reaches the rendezvous point first has to wait till the other task is ready to rendezvous as well. Thus there is synchronous communication, and the messages exchanged are guaranteed to be current, since senders cannot get arbitrarily ahead of receivers as would be possible in a buffered asynchronous communication scheme. However, it is also possible for a task to deadlock itself by waiting for a rendezvous that will never occur.

The syntax of a rendezvous is as follows: one task "calls" an entry declared in another. An "entry" is so named to signify the fact that, in a way, a task enters the body of another task. An entry is declared in the specification of a task as follows:

```
task Task_A is
     entry Entry_1 (<in, out or in out parameters>);
end Task_A;
```

The parameters are optional, i.e., there exist entries with no parameters. The syntax of the entry is very similar to that of a procedure call. In the body of the task there will be an "accept" statement corresponding to the entry statement. The syntax looks like this:

```
task body Task_A is
     ....    -- declarations;
begin
     ....
     accept Entry_1 (<formal parameters>) do
     ....    -- sequence of statements in
             -- accept body
     end Entry_1;
     ....
end Task_A;
```

The sequence of statements between the "accept Entry_1" statement and the "end Entry_1" is the "accept body" of Entry_1. It is the execution of the accept body that constitutes the rendezvous between Task_A and some other task that calls the entry Entry_1. The accept body is lexically part of the thread of control of the accepting task (here Task_A), and, in general, the semantics of the rendezvous can be visualized as meaning that the accepting task executes the accept body while the calling task (here Task_B) is blocked. We must caution that this is only a pedagogical convenience, and not necessarily a strict interpretation of the Language Reference Manual. In fact, there is nothing to prevent the alternate interpretation, that the accept body is executed by the calling task as part of its thread of control (while maintaining the accepting task's environment): One final note: the accept body can be null, in which case the rendezvous is merely a synchronization device, and does not pass any data.

An actual entry call looks very similar to a procedure call. To call entry Entry_1 in task Task_A, the calling task has to execute the statement

Task_A.Entry_1 (<parameters>);

thus giving the name of the called task as well. An interesting point to note is that not only tasks but procedures as well can make entry calls. The reason here is that every procedure is run as part of the thread of control of a task; thus it is the task executing the procedure that essentially makes the call. Furthermore, even for the procedure that conforms to the concept of the "main program", this argument is valid, since even this procedure is logically encapsulated by a hypothetical task.

An important point to note is that during the execution of the sequence of statements constituting the accept body, i.e. during the rendezvous, the two tasks are constrained to execute together. (Technically speaking, the tasks are synchronized only at the beginning and end of the rendezvous, but for purposes of understanding, we may think of them as actually executing together.) After the end of the rendezvous, they can continue on their way independently. This has interesting practical consequences -- the larger the sequence of statements in the accept body, the longer the blocked task (either the calling or accepting task -- see above) will have to wait. This means that there is very good reason to make the accept body as short as possible, since otherwise we are guilty of over-synchronization. Usually

the accept body consists merely of an assignment of the incoming data to local variables. This dogma is worth remembering: unless there is pressing need to do otherwise, do as little as possible in the accept body.

The form of the entry call is similar to a procedure call. The kinds of parameters allowed are in, out and in out parameters. An entry call made by Task_B to entry Entry_1 in Task_A will look like this:

```
task body Task_B is
     ...
begin
     ....     -- other statements
     Task_A.Entry_1 (<actual parameters>);
     ...      -- other statements
end Task_B;
```

Information flow is exactly as in the case of procedure calls. Entry parameters can be of mode in, out or in out, and the actuals have to match the formals as in the case of procedure calls.

From the above descriptions, it is evident that there is some similarity between procedure calls and rendezvous. However, the differences have to be highlighted. The most important difference is the blocking nature of the rendezvous. As mentioned above, the task which arrives first at the rendezvous point is blocked awaiting the arrival of the other task at the rendezvous point as well. This is quite different from the semantics of procedure calls which involves no waiting. If the procedure is implemented in reentrant fashion, several callers can call

it concurrently, and indeed execute the procedure body concurrently. However, exactly two tasks can be involved in executing a rendezvous at any given time. (This does not preclude the possibility of several tasks wishing to perform a rendezvous at the same time, i.e. making entry calls at the same time. In this case, they will be queued in FIFO order, and serviced as and when possible, as we shall see later.)

The description above gives only a small portion of the semantics o task communication. However, it is possible to model a real time system given just these constructs. Consider the following example taken from a real avionics system. There are three independent threads of control in the system, a front-end process which receives data from the external world, a main process which performs transformations on this incoming data, and a back-end process which controls communications with an output device. The front-end process accepts input from an operator or from a sensor and pre-processes it, perhaps by reformatting it. The main process performs numerical transformations on the data sent to it by the front-end. The back-end interfaces with a printer or other display. Note that in the examples which follow, processing statements will be written in a PDL style, but all tasking constructs will be legal Ada. The main process is in a loop performing essentially this:

```
loop    -- forever
     receive data from front-end;
     process data;
     send transformed data to back-end;
end loop;
```

The assumption may be made that the processing done by the tasks proceeds independently of each other; in other words, except when they actually need to communicate with each other to exchange data, the processes do not need to interact. Thus the most obvious Ada model would consist of three tasks, Front_End, Main_Task and Back_End. Thus the skeleton of the design consists simply of:

```
task Front_End is
end Front_End;

task Main_Task is
end Main_Task;

task Back_End is
end Back_End;
```

Of course, this merely declares three independent tasks (whose bodies are yet to be written). In order that they may communicate, we need to declare entries. Since input data goes from the Front_End to Main_Task, it seems most logical to include the data-exchange entry in Main_Task, so that Front_End can call it. Thus, we can flesh out some of the details of Front_End and Main_Task. The rendezvous entry is called Pass_Input_to_Main.

```
task Front_End is
end Front_End;

task Main_Task is
     entry Pass_Input_to_Main (<parameters>);
end Main_Task;
```

```
task body Front_End is
     ...
begin
     loop     -- forever
          receive data from operator/sensor;
          perform some operations on data;
          Main_Task.Pass_Input_to_Main (<parameters>);
          -- this is the rendezvous call
     end loop;
end Front_End;
```

The parameters of the entry call will consist of the input data, and to give it concreteness, let us assume that this is of type Raw_Data_Type. Similarly, data that has been processed by Main_Task has been converted to type Processed_Data_Type. The communication between Main_Task and Back_End can thus be modeled in analogous fashion by using the entry call to Send_Output to_Back_End. As we discussed earlier, the accept body should consist merely of assignments of incoming data to local variables. There should also be no confusion about the in mode of the parameters to the two entry calls. Thus we have the following:

```
type Raw_Data_Type is ...;
type Processed_Data_Type is ...;

task Front_End;    -- since it does not have an entry in it
                   -- we need not say ... is/ end Front_End;

task Main_Task is
     entry Pass_Input_to_Main (In_Data : in Raw_Data_Type);
end Main_Task;

task Back_End is
     entry Send_Output_to_Back_End
               (Out_Data : in Processed_Data_Type);
end Back_End;
```

```
task body Front_End is
     ...
begin
     loop   -- forever
         receive data from operator/sensor;
         perform some operations on data;

         Main_Task.Pass_Input_to_Main (<parameters>);
         -- this is the rendezvous call
     end loop;
end Front_End;

task body Main_Task is
     Local_Data : Raw_Data_Type;
     Out_Data   : Processed_Data_Type;
     ...
begin
     loop   -- forever
         accept Pass_Input_to_Main (In_Data : in Raw_Data_Type) do
              Local_Data := In_Data;
         end Pass_Input_to_Main;

         perform statements that transform Local_Data to Out_Data;

         Back_End.Send_Output_to_Back_End (Out_Data);
     end loop;
end Main_Task;

task body Back_End is
     Output_Data : Processed_Data_Type;
     ...
     begin
         loop   -- forever
              accept Send_Output_to_Back_End
                  (Out_Data : in Processed_Data_Type) do
                      Output_Data := Out_Data;
              end Send_Output_to_Back_End;

              send Output_Data to display device;
         end loop;
     end Back_End;
```

Furthermore, so that the types and entries are visible to each of the tasks, and to use the modularity inherent in Ada packages, let us assume that they are all included in the body of a package called Task_Package. This can be made a library package, for separate

compilation. The specification of the package does not have to include anything in it, since the body is self-contained, and there is no need for any external agency to enter the body. Thus the specification of Task_Package will not contain anything. An important point to be noted here, as explained in section 2.1, is that tasks that are elements of packages will be elaborated when the packages themselves are elaborated. Therefore, tasks that are included in library packages are active and ready to execute when the library package is imported by a main program. In our case, the main program is trivial, and merely prints out a message on the system console. The overall system looks like this now:

```
package Task_Package is
     -- nothing made visible
end Task_Package;

with Task_Package; use Task_Package;
with Text_IO; use Text_IO;
procedure Main is   -- "main program"
begin
     Put ("System is now active!!!");
end Main;

package body Task_Package is

     -- body of Task_Package: this is essentially the
     -- same as the previous code fragment

end Task_Package;
```

What we have now is essentially a complete Ada solution to the problem. The gaps in the pseudo-code have to be filled in with real Ada code, of course. Since much of the psuedo-code involves device interactions, it will be instructive to consider the facilities that Ada provides for this. There are two main Ada features, Low_Level_IO and interrupt handling, to deal with devices. The first, which is covered in more detail in a companion document (Advanced Ada) considers memory-mapped I/O and other interactions with memory.

Interrupt handling is provided by the underlying runtime support system, and frees the user from having to deal with the details of handling device interrupts. Interrupt handling is a subset of the features offered under task management. Essentially, the user can "attach" an entry to a memory location by using an "address clause" which specifies the location. Thereafter, the runtime support system treats a hardware interrupt occurring at that memory location as a hypothetical task making an entry call to the corresponding entry. The role of the traditional interrupt handler can thus be taken over by the accept body of the interrupt entry -- the sequence of statements there can perform the same actions. To give these ideas more concreteness, let us assume that the Front_End task receives device interrupts at location #16#0003AC78 which is Ada for the hexadecimal location 0003AC78. Let us also assume that all that it is necessary to do in the interrupt handler is to assign the incoming data to some local variable. Thus the declarations for handling the device interrupt (from say the keyboard device) would look like this:

```
task Front_End is
     entry Keyboard_Interrupt (Char_Data : in Character);
     for Keyboard_Interrupt use at #16#0003AC78;
     -- address clause
end Front_End;
```

A sensor interrupt would be handled in the same manner. In the body of Front_End, the statement "receive input from operator" can be translated into:

```
accept Keyboard_Interrupt (Char_Data : in Character) do
     Oper_Char := Char_Data;
end Keyboard_Interrupt;
```

where Oper_Char is a local variable of the appropriate type.

If the printer is an interrupting device, so that the Back_End process has to worry about Printer_Interrupts, a similar interrupt handler can be written for that as well. More details on the interrupt handling mechanism can be found in section 5.1.

The current version of our program, complete with the above embellishments, looks like the following:

```
package Task_Package is
     -- nothing made visible
end Task_Package;

with Task_Package; use Task_Package;
with Text_IO; use Text_IO;
procedure Main is       -- "main program"
     begin
         Put ("System is now active!!!");
end Main;
```

```
package body Task_Package is

     type Raw_Data_Type is ...;
     type Processed_Data_Type is ...;

     Keyboard_Interrupt_Loc : constant System.Address :=
                                   <interrupt location>;
     Sensor_Interrupt_Loc : constant System.Address :=
                                   <interrupt location>;
     task Front_End is
         entry Keyboard_Interrupt (In_Oper_Data : in Character);
         for Keyboard_Interrupt use at Keyboard_Interrupt_Loc;

         entry Sensor_Interrupt
                   (In_Sensor_Data : in <some data type>);
         for Sensor_Interrupt use at Sensor_Interrupt_Loc;
     end Front_End;

     task Main_Task is
         entry Pass_Input_to_Main (In_Data : in Raw_Data_Type);
     end Main_Task;

     task Back_End is
         entry Send_Output_to_Back_End
                  (Out_Data : in Processed_Data_Type);
     end Back_End;

     task body Front_End is
         Oper_Data   : Character;
         Sensor_Data : <some data type>;
     ...    -- local variables
     begin
         loop   -- forever
              accept Keyboard_Interrupt
                       (In_Oper_Data : in Character) do
                  Oper_Data := In_Oper_Data;
              end Keyboard_Interrupt;
```

```
                accept Sensor_Interrupt
                        (In_Sensor_Data : in <some data type>) do
                    Sensor_Data := In_Sensor_Data;
                end Sensor_Interrupt;

                perform some operations on data;

                Main_Task.Pass_Input_to_Main (<parameters>);
                -- this is the rendezvous call
            end loop;
        end Front_End;

        task body Main_Task is
            Local_Data : Raw_Data_Type;
            Out_Data   : Processed_Data_Type;
            ...
        begin
            loop   -- forever
                accept Pass_Input_to_Main
                           (In_Data : in Raw_Data_Type) do
                    Local_Data := In_Data;
                end Pass_Input_to_Main;

                perform statements that transform Local_Data to
                 Out_Data;

                Back_End.Send_Output_to_Back_End (Out_Data);

            end loop;
        end Main_Task;

        task body Back_End is
            Output_Data : Processed_Data_Type;
            ...
        begin
            loop   -- forever
                accept Send_Output_to_Back_End
                        (Out_Data : in Processed_Data_Type) do
                    Output_Data := Out_Data;
                end Send_Output_to_Back_End;

                send Output_Data to display device;
            end loop;
        end Back_End;

    end Task_Package;
```

The solution above is as complete as our limited view of Ada tasking will permit. It satisfies most of our requirements, but the astute reader may have noticed two serious shortcomings. These are a) the lack of non-determinism and b) oversynchronization. First of all, the structure of Front_End, wi' the two interrupt handlers for Operator_Interrupt and Sensor_Interrupt respectively, dictates the following sequence of events: operator interrupt, sensor interrupt, operator interrupt sensor interrupt .... This is the only sequence it can handle. Any other sequence will cause the Front_End to be blocked at one of its accept statements. Obviously, the requirement we have is more like this: if there is an operator interrupt, handle it; if there is a sensor interrupt, handle that. Thus we need a construct that is equivalent to our intuitive understanding of the "or" concept - accept either X or Y, as the case may be. This is provided by the "select" statement, which will be covered in detail later in this section. A solution containing selective wait statements will therefore be non-deterministic, i.e. it provides some flexibility to take care of events at execution time.

The second major problem with this solution is that of oversynchronization. This arises essentially because the tasks are very closely coupled, in the sense that they all rendezvous with each other directly. Because of the blocking nature of the rendezvous construct, this means that if any of the tasks is somewhat tardy, the whole system gets slowed down because of a ripple effect -- task A being slow delays

task B, which in turn delays task C, and so on. There is a simple solution to the problem of too close a coupling -- this involves the use of buffering tasks. This paradigm will be covered in detail in Exercise 4.1.

Without further ado, let us consider the syntax of the select statement. As we mentioned above, the rationale for this arises from the fact that decisions might have to be made at runtime about which rendezvous to accept, in cases where a variety of choices may sensibly be made. There are two kinds of select statements. One kind, called "selective wait," involves selecting among several accept statements. The second kind, called a "conditional entry call," involves one entry call and some choices. Conditional entry calls are less powerful than selective waits, and therefore we shall speak primarily of selective waits.

In the simplest case, a select statement is a choice among several possible rendezvous, and looks like the following. Note that the sequences of statements following the accept body are optional; however they may contain most kinds of statements.

```
select
     accept Entry_1 [(...)] [do
          ...
     end Entry_1];
     [sequence of statements_1;]
or
     accept Entry_2 [(...)] [do
          ...
     end Entry_2];
     [sequence of statements_2;]
or
     accept Entry_3 [(...)] [do
          ...
     end Entry_3];
     [sequence of statements_3;]
or
     ...
     -- and so on
end select;
```

Of course, there need only be two alternative accept statements to choose between. The semantics of the above statement are as follows: when an accepting task reaches the select statement, it may rendezvous with any of the entries in a non-deterministic fashion, i.e. the language does not define which rendezvous will be chosen. In particular, it is not necessarily the case that the rendezvous will be chosen in their textual order (i.e., in our example it is not true that Entry_1 will be attempted first, then Entry_2 and then Entry_3 and so on). Thus the select is not to be interpreted as "if possible, accept Entry_1; if not Entry_2; if not Entry_3 ...." The concept of "possible" has to be explained: if, when the accepting task arrives at an entry, some calling task has already made an entry call to it, then the rendezvous is considered immediately possible. Thus, the select statement simply guarantees that some one of the immediately possible rendezvous will be performed. If, on the other hand, none of the rendezvous is immediately

possible, then the accepting task will have to wait until some calling task calls one of the entries.

In order to apply this to our example system, consider again the structure of the Front_End task. If it is necessary that the Sensor_Interrupt or the Keyboard_Interrupt be handled, whichever becomes active, then the select statement can look something like this:

```
select
     accept Keyboard_Interrupt (...) do
         ....
     end Keyboard_Interrupt;
or
     accept Sensor_Interrupt (...) do
         ....
     end Sensor_Interrupt;
end select;
```

This solution does not impose any particular sequence on the interrupts coming into the system; it can handle them in whatever order they arrive.

An interesting twist to this situation is the case when we want to ignore certain rendezvous, depending on certain flags or "guards" which are attached to the branches of the select statement. Only when the guards are "open," i.e. only when they have the value True, will the corresponding rendezvous be considered eligible (regardless of whether they are immediately possible or not). As an instance of this, consider the situation in which we wish to ignore all Keyboard_Interrupts while certain critical Sensor_Interrupts are being handled. A guard which we

shall call Allow_Keyboard_Interrupt can be set and reset to allow or preclude recognition of Keyboard_Interrupts. (We shall not worry about what happens to interrupts if they are not handled). Thus the select statement would be enhanced as follows:

```
select
     when Allow_Keyboard_Interrupt =>
          accept Keyboard_Interrupt (...) do
          ...
          end Keyboard_Interrupt;

or
     accept Sensor_Interrupt (...) do
          ...
          if //critical sensor readings// then
               Allow_Keyboard_Interrupt := False;
               ...
          end if;
     end Sensor_Interrupt;
end select;
```

In the paragraphs above we have seen how the use of the select statement can solve the problems of oversynchronization and determinism. There are some other problems that are interesting. Consider for instance the situation where it is necessary to take some action based on time -- in our example system, it may be that some fault-tolerance is required. If the tasks are on different processors, and one of the processors crashes, the others should have some mechanism to avoid deadlock -- in the system as it is written, if any of the tasks dies a violent death, then the others will deadlock, since they are expecting a rendezvous, which will never happen. One of the simplest but effective steps against this possibility is to use a time-out. In this case, if the rendezvous does not happen within a rather generous time period,to be

determined by the individual run-time system, it is assumed that there has been an abnormal condition of some sort, and a different course of action is taken. Another example of this scenario in our example would be when the system is very lightly loaded -- i.e. messages are being received at a very slow pace, and it is desirable to use the main processing task to do some other useful work if its services are not required to process messages. In either of these cases, an upper limit can be placed on the time that a task will wait at a select statement. This can be done by means of the delay statement within the select statement. The semantics of this is that if none of the other branches of the select can be chosen before the delay expires, then the sequence of statements following the delay will be executed. A delay statement which performs the above may be as follows:

```
select
     accept Pass_Input_to_Main (In_Data : in Raw_Data_Type) do
          ...
     end Pass_Input_to_Main;

or

     delay 300.0;
     -- sequence of statements to be executed when the timeout
     -- has expired

end select;
```

In this case, if the rendezvous does not become possible in 5 minutes, the timeout sequence will be executed. The delay is expressed in seconds, and is a fixed point type defined in the predefined package Calendar. The delta for the type, Duration, is implementation-

dependent. It may be noted that there may be several delay statements in a select statement; they may also be guarded, just the same as accept statements.

Selective waits with delays can be used as a somewhat crude way of reducing oversynchronization in the absence of buffer tasks (to be discussed in depth in Exercises 4.1 and 4.2), at the cost of losing data or making incorrect assumptions. On the producer's side, if the consumer is very tardy, a timeout might be used to throw away the current data and start a new data acquisition cycle. On the other hand, if it is the producer that is being tardy, then the consumer may decide to stop waiting for new data, and to extrapolate from the previous batch of data. Both these alternatives are acceptable in certain situations, where the data is either not critical, or the current information can be deduced from previous information. The consumer's loop might look like this then:

```
loop          -- forever
     select
         accept Information (...) do
              ...
         end Information;
     or
         delay 120.0;
         -- perform extrapolations from previous data
     end select;
end loop;
```

Consider now the possibility of having a delay of 0.0 seconds, i.e. one in which the timeout alternative is executed if rendezvous with any of the accept alternatives is possible at once. This would be useful in a situation where there is critical processing that cannot be blocked. Ada provides an alternative to the select statement that has practically the same semantics as a delay of no time. This is the "else" alternative, which states that if none of the accept alternatives can be immediately rendezvoused with, the sequence of statements following the else is to be executed. In our example, consider the possibility that the processing task, Main_Task, is on a very heavily loaded processor, so that it should not be made to wait at any rendezvous. Then, its select statement with the Output_Buffer might be modified so that if the data transfer is not immediately possible, the Main_Task goes on to do some Background_Test. Thus, instead of a simple rendezvous call to the Output_Buffer like this,

```
Output_Buffer.Buffer_Output_Data (...);
```

we would have:

```
select
     Output_Buffer.Buffer_Output_Data (...);
else
     -- do Background tests
     Background_Test;
end select;
```

This would ensure that Main_Task is never blocked at the buffering point.

For the sake of completeness, it must be mentioned that there is another kind of select statement. In this case, there is at most one entry call in one of the branches. The other branch can be an "else" statement (leading to a conditional entry call) or a delay statement (leading to a timed entry call). In a conditional entry call, the rendezvous is performed if it can be done immediately. Otherwise the "else" alternative is chosen. In the timed entry call, the delayed alternative is chosen unless the rendezvous can be performed within the delay time specified. These are useful in some of the cases we have considered above: if the Front_End would prefer to lose the current data than be forced to wait more than a reasonable time, then its entry call to the Input_Buffer can be made into a timed entry call thus:

```
select
     Input_Buffer.Buffer_Input_Data (...);
or
     delay 180.0;
end select;
```

In this case, after waiting for three minutes, the Front_End decides to abandon the current data and to go on acquiring new data.

Now that the syntax of the select statement has been presented in considerable detail, it is time to talk about a most important issue -- that of termination. An especially astute reader would have noticed that there could be problems associated with the graceful shutdown of our little message-processing system -- what, for instance, happens when it is time to shut the whole system down? The termination conditions that

were introduced in section 2.1 imply that all the tasks in the system have to be terminated before the system can shut down. How does one terminate a running task? One of the ways of doing this will be to "abort" the task, which facility will be explained in section 5.1. This is a drastic step, however, and is not expected to be done lightly. It would be much more desirable if somehow the information that there was no more data to be expected, and that therefore it was time to shut down the system, could be propagated to all the tasks in the system. Thereupon, they would presumably terminate themselves in an orderly fashion. There is a simple way of doing this in Ada, and that is to include within the select statements an entry which explicitly signals the fact that the time has come to shut down. This information naturally has to arise at the Front_End, and propagate downstream. As each task accepts the "terminate-self" entry, it passes it on to the next task downstream and terminates itself by exiting from all loops etc., completing execution of its sequence of statements, and executing its end statement. Thus, eventually, the entire system would have shut down. To illustrate this, consider the following simplified version of our problem -- we revert back to only the Front_End, Main_Task and Back_End, and the details are not shown. Assume Front_End passes information to Main_Task via a select statement. The skeletons of the two tasks would then look like this:

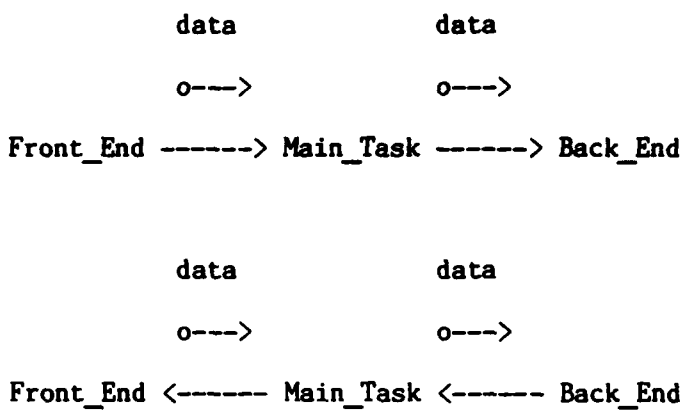
```
task body Front_End is
...
begin
     loop
          ...
          Main_Task. Pass_Input_to_Main (...);

          if <time to terminate> then
               Main_Task. Terminate_Self; -- termination entry
               exit;  -- leave the loop
          end if;
     end loop;
end Front_End;

task body Main_Task is
     ...
begin
     loop
          select
               accept Send_Input_to_Main (...) do
                    ...
               end Send_Input_to_Main;
          or
               accept Terminate_Self;
               exit;
          end select;

          ....

     end loop;
end Main_Task;
```

In this example, we have assumed that the exit from the loop is sufficient for termination. In cases that this is not so, it is of course possible to include the necessary statements following the acceptance of Terminate_Self. This style of explicit termination is simple, but might complicate the interfaces somewhat. For instance, when buffer tasks come into the picture, non-obvious measures have to be taken -- the standard generic buffers will have to be replaced by specially tailored buffers which can pass the entry along.

To pursue another approach, consider why it is that explicit termination is necessary -- this is because the tasks downstream are passive, and have no choice but to wait until the upstream task takes the initiative to hand it the data. Thus the downstream task will have no way of knowing if it is merely awaiting a tardy producer, or whether the producer has terminated already. If the producer has indeed terminated, then this scenario will produce deadlock in the consumer. However, consider what happens if the direction of the entry call is reversed -- the downstream consumer actively solicits data from the upstream producer. In this case, the rendezvous is driven by the consumer, who can thus become aware of the termination of the producer when it tries to rendezvous with the producer after the latter's death. Ada provides the user with a rather interesting rule regarding entry calls and exceptions. This states that if a task makes an entry call to an already terminated task, then the exception Tasking_Error will be raised in the caller. The caller will terminate, and the exception will not be propagated any further. This rule can be used to shut down the whole system gracefully if exceptionally. More on this device will be seen in section 5.1. Suffice to say here that merely reversing the direction of the entry calls can thus provide the unexpected bonus of graceful termination. To illustrate this briefly, consider our Front_End and Main_Task again. The code would now look like this:

```
task Front_End is
     entry Keyboard_Interrupt is ...
     entry Sensor_Interrupt is ...
     -- entries that are not relevant now

     entry Pass_Input_to_Main_Task
             (Input_Data : out Raw_Data_Type);

end Front_End;

task Main_Task is
     -- entries to deal with Back_End
     -- there are no entries to deal with Front_End
end Main_Task;

task body Front_End is
     ...
begin
     ....

     accept Pass_Input_to_Main_Task
               (Input_Data : out Raw_Data_Type) do
         Input_Data := <local data>
     end Pass_Input_to_Main_Task;

     ...
end Front_End;

task body Main_Task is
     ...
begin
     ...

     Front_End.Pass_Input_to_Main_Task (Local_Data);

     ...

end Main_Task;
```

Note the fact that the direction of the entry call has been reversed, and that the parameter is now an out parameter, as it should be. Pictorially, we can summarize the differences between these two designs:



The above approach to termination is not always feasible, since under some circumstances the entry call direction cannot be changed. Fortunately, Ada does provide a construct which works in most cases, and is in fact easier to use. However, the semantics of this statement are probably the most complicated part of tasking (and perhaps even the language), and it has to be used carefully. The construct in question is the selective wait with terminate alternative. This is essentially a select statement which has, in addition to one or more accept statements, a "terminate" statement. It may be noted that no delay alternatives and no else path is allowed in a select statement with a terminate alternative. The semantics of the statement is as follows: if a task is in a select statement where rendezvous is not possible with any of the accept alternatives, and the task's master has completed its execution, and all of the other dependents of the same master have either terminated or are similarly waiting at a select statement with a terminate alternative, then the master and all its dependents will terminate. The

meaning of this verbiage is simply that if all these conditions are satisfied there will never be any more rendezvous possible between the task in question, its siblings and its master, and therefore all of them will be effectively deadlocked, and might as well terminate since they cannot progress anymore. It follows from the definitions of task dependency and termination that no rendezvous are ever again possible with the task in question or the co-dependents of the task's master. In any case, the programmer's job is simplified to the extent of being a dogma -- use a terminate alternative in the select statements which perform data communication between producer and consumer. This will guarantee that when the producer is done, and there is no chance that there will be further data input, and all the other tasks in the system are also quiescent, then the entire system can terminate.

In our example system, consider the following: the select statement at the input side, between the Keyboard_Interrupt and the Sensor_Interrupt, has a terminate alternative associated with it; so do the select statements in Main_Task and Back_End which receive data from the upstream tasks. For simplicity, consider only the Front_End and Main_Task. Their code skeletons now look like:

```
task body Front_End is
     ...
begin
     ...
     select
          accept Keyboard_Interrupt (...) do
               ...
          end Keyboard_Interrupt;
     or
          accept Sensor_Interrupt (...) do
               ...
          end Sensor_Interrupt;
     or
          terminate;
     end select;
     ...
end Front_End;

task body Main_Task is
     ...
begin
     ...
     select
          accept Pass_Input_to_Main (...) do
               ...
          end Pass_Input_to_Main;
     or
          terminate;
     end select;
     ...
end Main_Task;
```

This will produce the desired termination properties.

Problem

The examples disussed in the tutorial illustrate the following tasking structures:

Front_End -----> Mai _ask -----> Back_End

Front_End <----- Main_Task <----- Back_End

The arrows represent the direction of entry calls. Notice that in the second case, they .o not mirror the direction of the data flow. There is another way to design this system in which the main task determines when it needs data and when it is ready to release the transformed data. Write this design in Ada and justify your design decisions.

Solution and Discussion

The design called for here has the following structure:

Front_End <----- Main_Task -----> Back_End

The complete code will be presented first, then its salient features discussed.

```
package Task_Package is
     -- empty package specification as before
end Task_Package;

with Task_Package;
with Text_IO; use Text_IO;
procedure Main is
begin
     Put ("The system is now active!!");

end Main;

with System;
package body Task_Package is
     type Raw_Data_Type is ... ;
     type Processed_Data_Type is ... ;
     type Sensor_Data_Type is ... ;

     Keyboard_Interrupt_Loc : constant System.Address :=
                              <interrupt location>;
     Sensor_Interrupt_Loc : constant System.Address :=
                              <interrupt location>;
     -- tasks which comprise the system are defined here.
```

```
task Front_End is
     entry Keyboard_Interrupt (In_Oper_Data : in Character);
     for Keyboard_Interrupt use at Keyboard_Interrupt_Loc;
     entry Sensor_Interrupt
             (In_Sensor_Data : in Sensor_Data_Type);
     for Sensor_Interrupt use at Sensor_Interrupt_Loc;
     entry Pass_Input_to_Main_Task
             (In_Data : out Raw_Data_Type);
     entry Terminate_Self;
end Front_End;

task Main_Task;  -- notice how there are no more entries
     --  declared in Main_Task:  it does all
     --  the calling but no more accepting.

task Back_End is
     entry Send_Output_to_Back_End
             (Out_Data : in Processed_Data_Type);
end Back_End;

task body Front_End is

     Allow_Keyboard_Interrupt : Boolean := True;
     Oper_Data                : Character;
     Sensor_Data              : Sensor_Data_Type;
     Raw_Data                 : Raw_Data_Type;
     ...  -- other local variables

begin    -- Front_End

     loop

         select
              when Allow_Keyboard_Interrupt =>
                  accept Keyboard_Interrupt
                          (In_Oper_Data : in Character) do
                      Oper_Data := In_Oper_Data;
                  end Keyboard_Interrupt;
                  <process keyboard input>;
                  Raw_Data := ...;

         or

              accept Sensor_Interrupt
                      (In_Sensor_Data : in Sensor_Data_Type) do
                  Sensor_Data := In_Sensor_Data;
              end Sensor_Interrupt;
              <process sensor input>;
              Raw_Data := ... ;
```

```
            or

                accept Pass_Input_to_Main_Task
                            (In_Data : out Raw_Data_Type) do
                    In_Data := Raw_Data;
                end Pass_Input_to_Main;

            or

                accept Terminate_Self;
                exit;

            end select;

        end loop;

    end Front_End;


    task body Main_Task is

        Shut_Down_Condition : Boolean := False;
        Local_Data          : Raw_Data_Type;
        Processed_Data      : Processed_Data_Type;
        -- other local variables

    begin     -- Main_Task

        loop

            ...

            Front_End.Pass_Input_to_Main_Task (Local_Data);

            statements to transform Local_Data to Processed_Data;

            Back_End.Send_Output_to_Back_End (Processed_Data);

            ...

            exit when Shut_Down_Condition;

        end loop;

        Back_End.Terminate_Self;

    end Main_Task;
```

```
task body Back_End is

     Processed_Data : Processed_Data_Type;
     ...   -- other local variables

begin   -- Back_End

     loop

          select

               accept Send_Output_to_Back_End
                       (Out_Data : in Processed_Data_Type) do
                   Processed_Data := Out_Data;
               end Send_Output_to_Back_End;
               send Processed_Data to display driver;

          or

               accept Terminate_Self;
               exit;

          end select;

     end loop;

end Back_End;

end Task_Package;
```

There are several interesting points about this solution. While at first glance, it appears very similar to the other solutions discussed in the tutorial, there is an important design difference. The placement of the entries and corresponding entry calls show a third way of building a tasking structure to model the front end, main task, back end message switching system. For any given tasking problem, the asymmetry of the tasking structure guarantees that there will be several equally correct though dissimilar solutions, depending on which tasks accept entry calls and which make entry calls. The differences in the structure are not intuitively obvious - they are ultimately reflected in the different

properties which characterize these solutions. For example, the direction of entry calls has a very definite impact on the tendency of a system to deadlock. In which tasks entries are declared also affects system performance. As was discussed in the tutorial, the direction of calls influences how the system can be gracefully shut down.

Looking at the specifics of this particular solution, the main task was designed without entries. Thus the main task assumes the "boss" role, receiving and sending data at its discretion rather than at the convenience of the front end or back end processors. This design provides for a buffering capability because the unprocessed data and the display data can be accumulated or displayed, as the case may be, without waiting for the main task to complete processing the data. Consider again the first solution given, in which the main task contains the entry Pass_Input_to_Main. Whenever the task Front_End is ready to send a data stream to the main task, it calls the appropriate entry: Main_Task.Pass_Input_to_Main. Having made this entry call, it must wait until the main task is available to accept this call. Assuming that this is a regular entry call as opposed to a timed entry call, the front end task risks slowing down the system as it is unable to handle either keyboard or sensor interrupts. Should zero data loss be a stringent requirement, in addition to performance requirements of no system thrashing, the front end task should be designed to continuously accept and accumulate new data.

As in the earlier designs shown, the front end task has entry declarations which handle both keyboard and sensor interrupts. Note the use of the address clause for these entries. Once again, this task consists of an infinite loop in which is embedded a selective wait. Provision is made for graceful system shutdown through the terminate alternative.

Similarly, the back end task is declared with an entry to accept processed data from the main task. Its design is analogous to the design of the front end task.

The main task effectively controls both the front end and the back end task insofar as it decrees when these tasks should die. The Boolean variable, Shut_Down_Condition, is local to the main task because policy decisions are supposed to be made by the main task and not by a data collection task such as the front end. Furthermore, if this switch were global, then the design would dictate the use of shared variables between tasks, a risky design practice. Should the operator issue a command to shut down the system, then this command will be transfered to the main task during the rendezvous, and the main task can act accordingly.

The moral of this exercise is twofold: the direction of entry calls and the direction of data flow represent two independent design decisions; and there is no single heuristic or design criterion to decide the direction of entry calls. In choosing the direction of entry calls,

the designer must consider the tradeoffs among performance objectives, system failure requirements, deadlock possibilities, and oversynchronization dangers, to name but a few.

EXERCISE 3.2

DEADLOCK

Objective

We will illustrate the style and power of Ada to solve classic deadlock problems and also illustrate problems and styles that are possible sources of deadlock.

Tutorial

The subject of deadlock arises in trying to achieve safety in program or operating system design. The design is constructed to keep processes from arbitrarily accessing code, shared data, and resources. A common practice is to use reentrant code to mutually access code and to use mutual exclusion for shared data and resources. The style and primitives used to achieve mutual exclusion can create deadlock, in which the processes block one another from execution.

Background

In multitask programs and multiprogram operating systems, data and resources are shared. If no restrictions are imposed, different processes access shared data simultaneously, giving unpredictable results. The restriction mechanism used to avoid this problem is mutual exclusion, i.e., allowing only one process at a time to access a shared resource. Detecting and reducing errors caused by mutual exclusion is the price paid for trying to protect data.

In a multiprogram operating system deadlock occurs when programs compete for the same resources. An example is as follows: a program A owns resource 1 and program B owns resource 2.

(A) <-- [1] (B) <-- [2]

Program A requests resource 2 but is blocked by program B, which has exclusive access to the resource, e.g. Program B is making use of resource 2.

(A) <-- [1] (B) <-- [2]

This is acceptable if, when program B has finished accessing resource 2, it releases resource 2. Suppose, however, program B also requires access to resource 1 prior to releasing resource 2. Now program A is still blocked by program B which is using resource 2, and program B is blocked by program A until A releases resource 1. Program A and B are deadlocked, each requiring a resource but suspended from execution. As other programs try to access resources 1 and 2 they too will be blocked; eventually all programs in the system will deadlock. If only one resource was accessed at a time, used, and released, deadlock would not occur. However, this style may not be practiced or workable for all applications. The operating system interface and design must try to arbitrate this resource allocation problem.

A single concurrent program may also have shared data and resources such as buses, disks, and pools of data. However a programming team can practice a style to minimize deadlock, and tools can be used to detect errors prone to causing deadlock.

Several solutions to achieve mutual exclusion have been developed. They include such approaches as busy wait, lock/unlock, semaphores, and rendezvous. We will use a simple producer to consumer process example to illustrate the traditional code solutions of busy wait, semaphore, and rendezvous.

BUSY WAIT

The overall scheme for sharing access to a resource with a busy wait is:

```
     <producer>                    <consumer>

loop                          loop
     .                             request resource;
     .                             wait if resource is in use;
     request resource;             use resource;
     wait if resource is           release resource;
       in use;
     use resource;                 .
     release resource;             .
end loop;                     end loop;
```

This overall structure is adequate, but if the consumer is in a tight loop and the producer is executing slowly, the consumer can release the shared resource and request it again before the producer has a chance to seize the resource. To avoid locking out one process in this

manner, the sharing algorithm needs to be sure the processes take turns using the resource. This solution is called Dekker's algorithm and may be expressed as follows:

```
<process>

-- request resource
Requested_By_Me := True;

-- wait if resource in use
while Requested_By_Other
loop
     if Last_User = Me then
     -- let other process have a turn
     -- wait for other process to finish
          while Last_User = Me
          loop
              null;  -- busy wait
          end loop;
     end if;
end loop;
-- use resource; -- critical section
Last_User := Me;
Requested_By_Me := False;
```

This solution was developed by the Dutch mathemetician Dekker. Only one process is presented because the paradigm is the same for both the producer and consumer processes. Each process explicitly passes the right to enter the critical section by setting its state to zero. When critical sections are lightly used, each process can enter the critical section immediately, because the other process is not using it. If the critical section is in use, then the most recent user waits by testing for completion of the other process. The variable Last_User is examined to decide which process is to proceed when both processes request entry. If the producer was the last user and the

processes are both now requesting access to the critical section, the producer executes the inner loop and performs a busy wait, providing fairness. If the consumer gets into the critical section before the producer arrives, then the producer stays in the outer loop performing a busy wait. This approach ensures mutual exclusion by ensuring no other processing may occur until the critical section is left.

SEMAPHORE APPROACH

```
     <producer>                <consumer>

loop                        loop
     .                         Wait(Semaphore);
     .                         -- critical section
     Wait(Semaphore);          Signal(Semaphore);
     -- critical section       .
     Signal(Semaphore);        .
end loop;                   end loop;
```

The wait primitive checks if the semaphore is set. If not set, the Wait routine sets it and returns, allowing entry into the critical section. If the semaphore is set, the calling process is blocked. The Signal primitive checks to see if a process is blocked by the semaphore and unblocks the process. If no process is blocked then the semaphore is cleared. This approach insures mutual exclusion by creating guards to the critical section.

The operating system definitions of semaphore and primitives may be as follows:

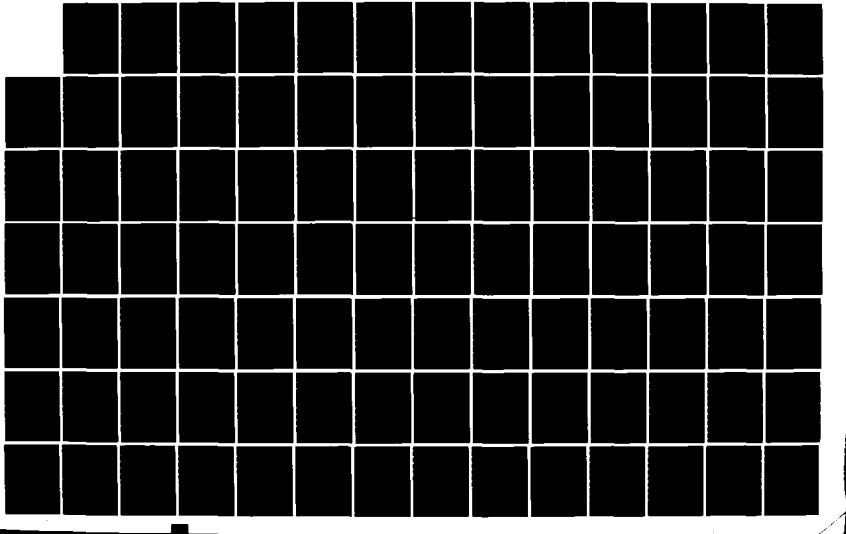
```
package Semaphores is

     type Semaphore_Type is private;
     procedure Wait (Semaphore: Semaphore_Type);
     procedure Signal (Semaphore: Semaphore_Type);

private

     type Semaphore_Type is
          record
               State : Boolean := True;
          end record;
     -- ensures appropriate initial value

end Semaphores;

package body Semaphores is

     Stop : constant Semaphore_Type := (State => False);
     Go   : constant Semaphore_Type := (State => True);

     procedure Wait (Semaphore : Semaphore_Type) is
     begin

          if Semaphore = Go then
               Semaphore := Stop;
          else
               Block_Calling_Process;
          end if;

     end Wait;

     procedure Signal (Semaphore : Integer) is
     begin
          if Process_Waiting then
               Unblock_a_Process; -- allow entry to critical
                                  -- section
          else -- no one wants the critical section
               Semaphore := Go;
          end if;
     end Signal;
end Semaphores;
```

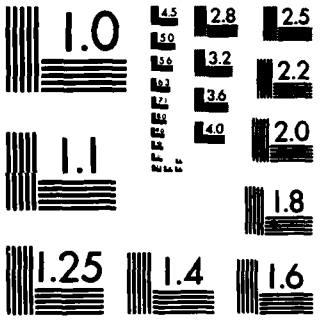
A user declares a semaphore variable to guard the critical section. (The use of record variable ensures that each semaphore has the correct initial value.) Each process that wants to access the

AD-A146 258 REAL-TIME ADA (TRADEMARK)(U) SOFTECH INC WALTHAM MA
JUL 84 DAAB07-83-C-K514

2/4

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

critical section must check the state of the semaphore prior to entering the section and must signal when it has finished executing the critical section. The use of semaphores guarantees that only one process will enter the critical section. When more than one process is attempting to enter the critical section, all but one of the processes is blocked. The procedure Unblock_a_Process determines which blocked process will next be allowed to enter the critical section after the currently executing process signals that it has finished.

The action of testing the state of the semaphore must be itself performed in a critical section, i.e., at most one process can be allowed to check, and possibly set, the state of the semaphore. Otherwise, two processes could independently call Wait and discover that the semaphore is in the Go state; both would be allowed to enter the critical section. The correct operation of Wait is guaranteed if the Wait procedure is executed by at most one process at a time.

Although the use of semaphore is not generally the best way to control access to shared resources in Ada, they are sometimes appropriate. In such cases, they should be implemented by a task. An appropriate Ada implementation of semaphores would be the following:

```
generic
package Semaphores is

     procedure Wait;
     procedure Signal;

end Semaphores;

package body Semaphores is

     task Sema is

          entry Wait;
          entry Signal;

     end Sema;

     task body Sema is
     begin
          loop
               accept Wait;
               accept Signal;
          end loop;
     end Sema;

     procedure Wait is
     begin
          Sema.Wait;
     end Wait;

     procedure Signal is
     begin
          Sema.Signal;
     end Signal;

end Semaphores;
```

A semaphore is created for each instantiation of the package. Calling the instantiated package's Wait and Signal procedures is equivalent to calling the Wait and Signal entries of the Sema task. If no processes are currently executing in the critical section, then the call to Sema.Wait will be accepted, and the Wait procedure will

return allowing the process to enter the critical section. If another process then attempts to enter the critical section, the call to Sema.Wait will be blocked, since the Sema task will be waiting to accept a call to the Signal entry. Processes are thus automatically queued on the Wait entry of the Sema task. When Signal is called, the Sema task loops to the Wait entry, and accepts a call from a waiting task, if there is one.

Note that if any process fails to call Signal, deadlock will occur, since Sema will wait forever at the Signal accept statement.

RENDEZVOUS

The Ada rendezvous can be used to enforce mutually exclusive access to some resource. The consumer process could be a task exporting the entry Transfer as follows:

```
task Consumer is
     entry Transfer (Data : Data_Type);
end Consumer;
```

and would then have in the body

```
task body Consumer is
     Local_Data : Data_Type;
begin
     loop
          accept Transfer (Data : Data_Type) do
               Local_Data := Data; -- copy to local storage
          end Transfer;
          -- process data
     end loop;
end Consumer;
```

The producer process could then make an entry call directly to Consumer

```
task body Producer is
     Data : Data_Type;
begin
     loop
          -- generate Data
          Consumer.Transfer(Data);
     end loop;
end Producer;
```

Note that semaphores were not needed. If there are several Producer processes, each must wait its turn to pass data to the Consumer task. The rendezvous implementation using accepts and entry calls provides a clean synchronization and message passing mechanism.

Deadlock Sources

Sources of deadlock for multiuser and multiprocess systems can differ due to the deterministic nature of the number of processes and mutual exclusion calls of the two systems.

An example of resource management (a disk drive) may be used to illustrate that, even if there are no semantic errors associated with a mutual exclusion coding strategy, there can be errors resulting in deadlock. Assume that a disk drive is shared by two processes and both generate more data than they consume from the disk drive. As time goes on, the disk becomes full. Now both processes are blocked because they have exhausted the disk space. Although semaphores or other

mutual exclusion mechanisms are used, the resource is depleted and the processes starve, creating deadlock.

Another example is when two processes have different priority and they can only be blocked by accessing a shared disk drive. The process of lower priority could be blocked indefinitely by the higher priority process. Thus the low priority task suffers lockout. (This is analogous to the dining philosophers problem discussed in Exercise 1.1.)

The above examples illustrate cases of multiuser systems that have an indeterminate number of users making an indeterminate number of resource requests upon a finite number of resources. For these cases we have shown that user processes can block each other by using mutual exclusion and by allowing access to another resource during execution of a critical region. We have also illustrated how resource exhaustion can produce deadlock even though mutual exclusion protects the resource.

Multiprocess programs may have a determinate number of processes making a determinate number of resource requests upon a finite amount of resources. In these cases, design errors can cause processes to be thrust into deadlock or lockout. A system can have higher priority processes lockout processes with lower priorities. Errors in design can be caused by an improper number of semaphore primitive calls or

where an improper sequence of semaphore calls are used. An example of a design error is when spurious calls to the semaphore primitives are made. When the

```
Wait(S);
Critical_Section;
Signal(S);
```

paradigm is not used, deadlock occurs.

Another area of errors arises when the program changes modes. A transition is made from one set of active processes to another. The new processes were asleep in the previous mode. Blocked processes on queues from the previous mode have to be cleared and put to sleep. Semaphores have to be properly initialized for the processes of the new mode, including the processes that continue to be active in the new mode.

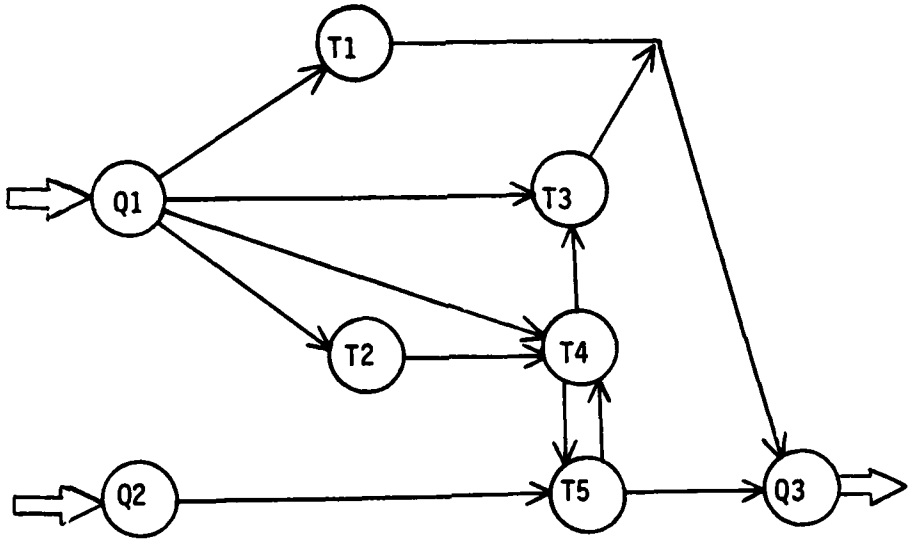
Pool Design

When a programmer is faced with implementing a set of communicating processes the possiblity of deadlock is always present. Let us look at such a case and illustrate the danger of deadlock. We will develop Pools to avoid such dangers.

A pool is simply a repository for information. Any item in the pool is available to any process that uses the pool. Instead of direct rendezvous for the purposes of synchronization, two tasks read and write data in the pool to effect communication. Using a pool has

the effect of reducing coupling among processes. The pool concept is particularly useful when there are processes that need to operate continuously but do not necessarily need updated information on every iteration. A pool is useful in a system in which many processes share the same data and the direct communication of all processes via rendezvous is too confusing or too inefficient.

From an avionics navigation design a data flow graph is developed below where the nodes are processes and the arcs (arrows in the following diagram), are data flow paths. Note that processes T4 and T5 have a cycle, i.e. where one can follow the directed arcs from T4 to T5 back to T4. Processes Q1, T2, and T4; and Q1, T4, and T3 produce subgraphs that could be written such as to produce deadlock.



An example of deadlock could be:

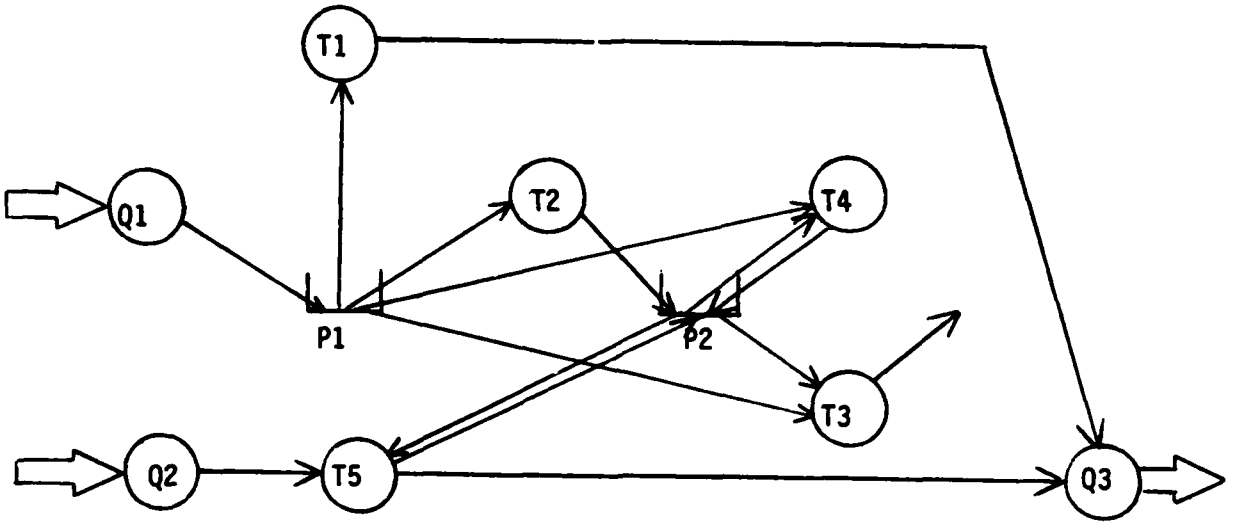
```
<Q1>            <T2>            <T4>
.
.
T4.Entry_1      accept Entry_2  accept Entry_3
T2.Entry_2      T4.Entry_3      accept Entry_1
.               .               .
end Q1;         end T2;         end T4;
```

Task Q1 waits on T4.Entry_1, Task T2 waits on call for Entry_2 (Q1), Task T4 waits on call for Entry_3 (T2). Deadlock!

The processes where the large arrows are directed into the nodes are hardware interface processes that receive external data. The nodes where the large arrow exits are processes that send data externally. The processes have the following communications

	# Rendezvous		
Process	In	Out	Total
Q1	<1>	4	5
Q2	<1>	1	2
Q3	2	<1>	3
T1	1	1	2
T2	1	1	2
T3	2	1	3
T4	3	2	5
T5	2	2	4

The interprocess communication would require 12 rendezvous (number of arcs) with processes T4 and T5 having cyclic coupling. Let us build 2 pools for this data flow process graph. Because Q1 has 4 Out rendezvous and an external rendezvous, let one Pool serve processes Q1, T1, T2, T3, and T4. Then let processes Q2, Q3, T4, and T5 be served by a second pool. The revised data flow graph is:



Note that T4 uses both pools because it needs data items in both of them. One pool could have been used, but using two pools provides more control over access to data and provides posting and retrieval of information in either pool in parallel. The processes now have the following communications:

	# Rendezvous		
Process	In	Out	Total
P1	1	4	5
P2	3	3	6
Q1	<1>	1	2
Q2	<1>	1	2
Q3	2	<1>	3
T1	1	1	2
T2	1	1	2
T3	2	1	3
T4	2	1	3
T5	2	2	4

The interprocess communication is now 3 with 11 pool communications for a total of 14 rendezvous. The pool processes may be constructed as

tasks with selective waits having 5 and 6 rendezvous in pools P1 and P2 respectively. The pools can have only a Get_Data and Put_Data entry to service the different calls. This allows pool execution if any rendezvous is possible while the other processes are blocked until the pool process services them.

The P2 pool design and T2 task are developed below:

```
package Pool_P2 is

     procedure Read (Data : out Data_Type);
     procedure Write (Data : in Data_Type);

end Pool_P2;

package body Pool_P2 is

     task Manager is
          entry Get (Data : out Data_Type);
          entry Put (Data : in Data_Type);
     end Manager;

     procedure Read (Data : out Data_Type) is
     begin
          Manager.Get (Data);
     end Read;

     procedure Write (Data : in Data_Type);
     begin
          Manager.Put (Data);
     end Write;
```

```
        task body Manager is

            Pool_Data : Data_Type := ... ;

        begin
            loop
                select
                    accept Get (Data : out Data_Type) do
                        Data := Pool_Data;
                    end Get;
                or
                    accept Put (Data : in Data_Type) do
                        Pool_Data := Data;
                    end Put;
                end select;
            end loop;
        end Manager;

    end Pool_P2;
```

The pool exports two procedures to put and get data into and out of the pool respectively. The procedures then simply call the corresponding entry in the pool manager as follows:

```
    with Timing, Pool_P1, Pool_P2;
    use Timing; -- This package provides the function Until
                --  a routine to yield cyclic execution
    separate (Avionics_Package)
    task body Task_T2 is

    begin
        loop

        --cyclic sampler
            delay Until (frequency_32_Hz);

            Pool_P1.Read (Data_1 : Data_Type_1);

            Compute_INS_State;

            Pool_P2.Write (Data_2 : Data_Type_2);
        end loop;

    end Task_T2;
```

The task T2 communicates with the pools via simple procedure calls for read and write from the task.

Let us consider another example of task activity control. A radar sweeps over n sectors in a single scan. After each sector is scanned, the blips detected in it must be processed before this sector is swept by the next radar scan. Each sector undergoes identical processing, and each sector can be processed asynchronously. Given these considerations, it has been decided to model the processing for each sector by a single task.

The problem may be stated in the following terms: given the task per sector representation, how does one express the requirement that the processing for a given sector must be completed in the time the radar takes to sweep n-1 sectors. If the processing is not completed in this time, it must be stopped. The processing task is then restarted with the fresh data from the new radar sweep of the sector.

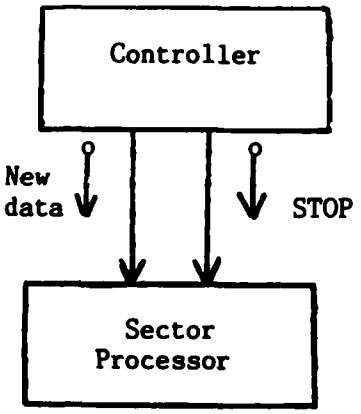
In order to stop the sector processing task after the radar has scanned n-1 sectors, one could abort that task. This solution, however, is dangerous because the task might be aborted at an awkward moment, e.g., while it is updating a database.
Upon closer analysis, an equivalent statement is as follows: after the radar has scanned n-1 sectors, the sector processing task is

preempted and resumes when fresh data is available. A PDL statement of the above is straightforward:

```
loop
     get new data for sector;
     while blips to process and not preempted
     loop
          process a blip;
          check for preemption;
     end loop;
end loop;
```

One way to preempt the sector blip processing task is to use a selective wait whose alternatives are to accept a preemption command from the controlling task or to resume processing a blip. Care must be taken when using this technique to avoid creating a situation where either deadlock might occur or old data might be reprocessed. Both these possibilities are discussed in greater detail below.

The basic mechanism consists of a sector processing task with two entries. One entry allows this task to get the radar data and the other entry allows this task to be preempted should its processing time have run out. The structure chart is shown below:



An initial implementation of the PDL could look like this:

```
task type Sector_Processing_Task is   -- this task is
                                      -- embedded inside
                                      -- a controller
                                      -- task, shown
                                      -- below, in which
                                      -- the type
                                      -- declaration for
                                      -- Blips occurs.
     entry New_Data (Sector_Data : Blips);
     entry Stop;
end Sector_Processing_Task;

task body Sector_Processing_Task is
     Local_Data : ... ;
begin
     loop
          accept New_Data (Sector_Data : Blips) do
              Local_Data := Sector_Data;
          end New_Data;
          while Blips_To_Process
          loop
              Process_Blip;
              select                  -- If preempt call has
                  accept Stop;        -- occurred then accept
                  -- exit loop        -- it, exit the blip
                  exit;               -- processing loop and
              else                    -- be ready to accept
                  null;               -- fresh data, or else
              end select;             -- continue to loop through
          end loop;                   -- and process the available
     end loop;                        -- blip
end Sector_Processing_Task;
```

The relevant part of the controller task would look like:

```
task Sector_Processing_Controller is
     entry New_Sector_Pulse;
end Controller;
```

```
task body Sector_Processing_Controller is

     type Sector is range 0 .. 15;
     subtype Blips_Allowed is Integer range 0 .. Max_Blips;
     type Blips (Num_Blips : Blips Allowed) is
          record
              Data : Blip_Data(1 .. Num_Blips);
          end record;

     ...
     type Sector_Processors is array
              (Sector'First .. Sector'Last)
           of Sector_Processing_Task;
     Sector_Processing : Sector_Processors;
     ...
begin    -- Sector_Processing_Controller
     ...
     loop
       -- Give new data for this sector.  Assume Sector_Data
       --  is hardware updated.
          Sector_Processing(Current_Sec).
                          New_Data(Sector_Data);
       -- Stop next sector's processing
          Sector_Processing((Current_Sec mod
                          Sector_Processors'Length)+1).Stop;
                  -- Assume the array index starts at 1;
          accept New_Sector_Pulse;  -- Synchronize with radar
     end loop;
     ...
end Sector_Processing_Controller;
```

An analysis of this solution reveals the following flaw. Consider the case that there is sufficient time to process all the blips, so that the sector processing task is now waiting to accept new data. The controlling task will issue its stop call after the radar has scanned n-1 sectors. The tasks are deadlocked because the sector processing task cannot proceed until it receives a New_Data call, and the controller task cannot proceed until its Stop call is accepted.

One way to eliminate the deadlock is to put the New_Data inside a selective wait whose other alternative is to accept a Stop call. The relevant part of this solution follows:

```
loop
     select
          accept New_Data (Sector_Data : Blips) do
               Local_Data := Sector_Data;
          end New_Data;
     or
          accept Stop;
     end select;
     while Blips_To_Process
     loop
          Process_Blip;
          select
               accept Stop;
     -- exit loop
               exit;
          else
               null;
          end select;
     end loop;
end loop;
```

This solution introduces a new problem, however, in that the old data gets processed anew, instead of the fresh data being received. Indeed one does not miss the Stop call, but as soon as the preemption is acknowledged, the flow of control drops through to the inner loop and the old data is reprocessed. The new data is not received until all the old data is reprocessed; the controller task hangs until the new data is accepted and in short, the entire processing goes haywire.

Both these problems can be avoided by introducing another control statement after the blip processing loop. If there are no more blips

to process then the sector processing task waits until it receives a Stop call. This solution is shown in the following section. It is interesting to note that at the PDL level this extra control statement is not visible; it becomes necessary at the implementation level.

```
task Sector_Processing_Controller is
     entry New_Sector_Pulse;
end Sector_Processing_Controller;

task body Sector_Processing_Controller is

     type Sector is range 0 .. 15;

     subtype Blips_Allowed is Integer range 0 .. Max_Blips;
     type Blips (Num_Blips : Blips_Allowed) is
          record
               Data : Blip_Data (1 .. Num_Blips);
          end record;

     task type Sector_Processing_Task is
          entry New_Data (Sector_Data : Blips);
          entry Stop;
     end Sector_Processing_Task;

     type Sector_Processors is array (Sector'First .. Sector'Last)
                                      of Sector_Processing_Task;

     Sector_Processing:  Sector_Processors;
```

```
     task body Sector_Processing_Task is

          Local_Data : Blips;
          Preempted  : Boolean;
          procedure Process_Blip (A_Blip : Blip_Data_Element) is separate;

     begin   -- Sector_Processing_Task
          loop
               accept New_Data (Sector_Data : Blips) do
                    Local_Data := Sector_Data;
               end New_Data;
               Preempted := False;
               for Curr_Blip in 1 .. Local_Data.Num_Blips loop
                    Process_Blip (Local_Data.Data(Curr_Blip));
                    select
                        accept Stop;
                        Preempted := True;
                    else
                        null;
                    end select;
                    exit when Preempted;
               end loop;
               if not Preempted then
                    accept Stop;
               end if;
          end loop;

     end Sector_Processing_Task;

begin     -- Sector_Processing_Controller
     ...
     loop

          for Current_Sector in Sector'Range loop
               accept New_Sector_Pulse;
               Sector_Processing((Current_Sector mod Number_Of_Sectors)+1).Stop;
                                 -- Stop following sector's processing
               -- Assume Sector_Data is hardware updated
               Sector_Processing(Current_Sector).New_Data(Sector_Data);
          end loop;
     ...
     end loop;

end Sector_Processing_Controller;
```

This example illustrates a way of suspending a task and letting it resume. Ada tasking does not provide for regularly starting and stopping tasks. For applications which seem to need such a capability, the designer should express this requirement using one of the tasking constructs discussed.

Problem

As we have designed Sector_Processing.Controller, will it ever start execution? Hint: look for deadlock conditions upon startup. If deadlock occurs propose a solution.

Discussion and Solution

No. The Sector Processing tasks will first be blocked at New_Data while the Sector_Processing_Controller will attempt to rendezvous at New_Sector_Pulse and then at Sector_Processing(i).Stop. Since the Sector_Processing tasks are at New_Data, the conclusion is that deadlock occurs upon startup.

The Sector_Processing_Controller and Sector_Processing should be initially synchronized in the operational loop. There are several ways to correct this deadlock condition. One is to synchronize the first sector in the Controller before the execution loop and provide an

```
accept Stop;
```

outside the execution loop in the task body Sector_Processing_Task. The first sector task and controller now enter the execution loop. Upon reaching the stop entry, the controller increments the sector task modulous + 1 and releases the sector task from the initial synchronization call of each sector task. Therefore, all remaining sectors in order enter the execution loop. The following code details this solution:

```
task Sector_Processing_Controller is
     entry New_Sector_Pulse;
end Sector_Processing_Controller;
```

```
task body Sector_Processing_Controller is

     type Sector is range 0 .. 15;
     subtype Blips_Allowed is Integer range 0 .. Max_Blips;
     type Blips (Num_Blips : Blips_Allowed) is
          record
               Data : Blip_Data (1 .. Num_Blips);
          end record;

     task type Sector_Processing_Task is
          entry New_Data (Sector_Data : Blips);
          entry Stop;
     end Sector_Processing_Task;

     type Sector_Processors is array (Sector'First .. Sector'Last)
                                      of Sector_Processing_Task;

     Sector_Processing:  Sector_Processors;

     task body Sector_Processing_Task is

          Local_Data :  Blips;
          Preempted  :  Boolean;
          procedure Process_Blip (A_Blip : Blip_Data_Element) is separate;

     begin     -- Sector_Processing_Task

          accept Stop;  -- to initialize startup
          loop
               accept New_Data (Sector_Data : Blips) do
                    Local_Data := Sector_Data;
               end New_Data;
               Preempted := False;
               for Curr_Blip in 1 .. Local_Data.Num_Blips
               loop
                    Process_Blip (Local_Data.Data(Curr_Blip));
                    select
                        accept Stop;
                        Preempted := True;
                    else
                        null;
                    end select;
                    exit when Preempted;
               end loop;
               if not Preempted then
                    accept Stop;
               end if;
          end loop;
     end Sector_Processing_Task;
```

```
begin  -- Sector_Processing_Controller

     ...

     Sector_Processing(Sector'First).Stop; -- initialize first sector

     loop

          for Current_Sector in Sector'Range
          loop
               accept New_Sector_Pulse;
               Sector_Processing
                    ((Current_Sector mod Number_Of_Sectors)+1).Stop;
                              -- Stop following sector's processing
               -- Assume Sector_Data is hardware updated
               Sector_Processing(Current_Sector).New_Data(Sector_Data);
          end loop;

          ...

     end loop;

end Sector_Processing_Controller;
```

Another solution is to use

```
          Preempted := False;
```

instead of the accept Stop; statement in the Sector_Processing_Task

and move the check for Preemption

```
          if not Preempted then
               accept Stop;
          end if;
```

from the bottom of the loop to the top. This will now synchronize the

Sector_Processing_Task task units.

CHAPTER 4

FUNDAMENTAL TASK DESIGNS

EXERCISE 4.1

FUNDAMENTAL TASK DESIGNS

Objective

This section is designed to develop the message buffer paradigm introduced in Section 3.1 as a means of reducing the coupling of two tasks. Buffering is a technique in wide use to reduce over synchronization and to improve system throughput.

Tutorial

We have seen how cooperation between tasks is achieved using Ada. We have seen that the Ada mechanism of rendezvous is used to effect task synchronization. The example of the front end - process - back end system developed in Section 3.1 illustrated the design of a real-time system using Ada and pointed out the options available for an Ada implementation of such a system. One of the major options makes use of a message buffer to hold data to be passed between tasks instead of direct rendezvous. The advantages to buffering are reduction of coupling between tasks and possibly improved system throughput.

The tasks in this example are closely coupled in that they rendezvous directly with each other. At any entry call or accept statement a task will wait until a rendezvous is possible. When a producer has data available, it must wait until a consumer requests it before continuing the process of producing the next value. In a real-

time situation, a delay in rendezvous by reason of the consumer being slowed down for some reason can result in loss of input data. Similarly, when a consumer needs data, it must wait until a producer is ready to rendezvous. If the producer is delayed then he consumer may get behind in its processing, unable, for example, to extrapolate input since it is waiting for a rendezvous.

One technique that is widely used to handle such timing problems is buffering. Buffering makes use of a storage area in which data is posted and retrieved under a set protocol, but in which posting and retrieving are not necessarily performed in an alternating order. Let's look at an implementation of buffering in Ada. We will use a generic package because it would be useful to allow any type of data to be buffered.

```
generic

     Bufmax : Natural;
     type Item is private;

package Buffer is

     procedure Put (Data : in Item);

     procedure Get (Data : out Item);

end Buffer;
```

Two things, Bufmax and Item, must be specified when the package is instantiated. Item is the type of the object to be buffered while Bufmax is the maximum number of items that will be buffered. For example, the instantiation

```
package Char_Buffer is new Buffer (Bufmax => 16, Item => Character);
```

will create a package with two procedures, Char_Buffer.Put and Char_Buffer.Get, that will buffer up to sixteen characters. The Put procedure will post a character in the buffer and the Get procedure will retrieve one of the characters posted in a first-in-first-out (FIFO) sequence. What happens if Put is called Bufmax+1 times in a row before Get is called? There are several options here, but let's define this situation to be handled in such a way that Put will wait for a Get to retrieve a data item. Similarly, if Get is invoked when the buffer is empty, then it will wait until another data item is Put. Note that if this package is instantiated with a Bufmax of 1 then the way we have defined the operation of Put and Get yields almost the same behavior as using direct rendezvous.

Now let's consider the implementation of this package. We will use an array of Bufmax Items to act as the buffer. We will maintain two index values in the array, the index of the component to Get and the index of the next component to Put. Furthermore, we will maintain the count of the number of items in the buffer. Since Put and Get may be called in any order (and in fact can be invoked simultaneously!), we will have to control carefully the placement of data in the array. In Ada, we will use a task to synchronize the operations of putting an item into and getting an item out of our array. Let's take a look at a body for package Buffer.

```
package body Buffer is

     task Buffer_Mgr is
          entry Write (Data : in Item);
          entry Read  (Data : out Item);
     end Buffer_Mgr;

     task body Buffer_Mgr is

          Buffer_Area : array (1 .. Bufmax) of Item;
          Count       : Integer range 0 .. Bufmax := 0;
          Next_Read   : Integer range 1 .. Bufmax := 1;
          Next_Write  : Integer range 1 .. Bufmax := 1;

     begin -- Buffer_Mgr
          loop
               select
                    when Count < Bufmax =>
                    -- there is still space in the array.
                    accept Write (Data : in Item) do
                      Buffer_Area (Next_Write) := Data;
                    end Write;
                    Next_Write := (Next_Write mod Bufmax) + 1;
                    Count := Count + 1;
               or
                    when Count > 0 =>
                    -- there is still data to be read.
                    accept Read (Data : out Item) do
                      Data := Buffer_Area (Next_Read);
                    end Read;
                    Next_Read := (Next_Read mod Bufmax) + 1;
                    Count := Count - 1;
               end select;
          end loop;
     end Buffer_Mgr;

     procedure Put (Data : in Item) is
     begin
          Buffer_Mgr.Write (Data);
     end Put;

     procedure Get (Data : out Item) is
     begin
          Buffer_Mgr.Read (Data);
     end Get;

end Buffer;
```

The task Buffer_Mgr manages the reading and writing of data from the buffer which is the array Buffer_Area. The value of Count is the number of components in Buffer_Area that contain data. When Count is zero, there is no data to be read and thus the select alternative containing the accept statement for Read is closed, making any calls to Get wait for data. When Count is Bufmax, then the buffer is full, and thus the select alternative containing the accept statement for Write is closed, making any calls to Put wait until space is available. Note that Ada guarantees that when the buffer is empty (i.e. Count = 0), calls to Read are queued in the order that they are made. Similarly, when the buffer is full, calls to Put will be queued in the order that they are made. It follows then that our implementation has the desirable property that the Nth Get always yields the data placed in the buffer by the Nth Put.

Notice that Get and Put are implemented via a single Task having two entries. Why have two procedures been made visible from this package instead of this single task? The answer lies in a design principle that exposes no more detail of implementation than is absolutely required. It is really not essential for a user of package Buffer to know that a task is used to implement the two operations of posting and retrieving data. That same design principle places the definition of Buffer_Area within the task instead of in the package body. The fact that an array is used is a detail of the way that task Buffer_Mgr works. Procedures Get and Put do not need these details.

We now have a package that implements buffering as we have chosen to define it. However, in the description given above, it is not quite correct to use the word "wait" to describe the Put operation on a full buffer or a Get operation on an empty buffer, since we really derived the term from our implementation using tasks. Let's rephrase the descriptions of this package as, "This package is to be instantiated with an appropriate maximum buffer size and data item type so as to provide buffering capabilities. The procedure Put stores a data item in the buffer and the procedure Get retrieves a data item from the buffer such that the Nth item stored is the Nth item retrieved."

Now let's look at how we might use this package in the context of the example in Section 3.1. Consider the following definition of the body for package Task_Package.

```
package body Task_Package is

     type Raw_Data_Type is ...;
     type Processed_Data_Type is ...;
     Keyboard_Interrupt_Loc:  constant System.Address :=
                              <interrupt location>;
     Sensor_Interrupt_Loc  :  constant System.Address :=
                              <interrupt location>;

     task Front_End is
          entry Keyboard_Interrupt (In_Oper_Data : in Character);
          for Keyboard Interrupt use at Keyboard_Interrupt_Loc;
          entry Sensor_Interrupt
               (In_Sensor_Data : in <some data type>);
          for Sensor_Interrupt use at Sensor_Interrupt_Loc;
          entry Terminate_Self;
     end Front_End;
```

```
task Main_Task is
     -- no longer need an entry to receive data from Front End
     entry Terminate_Self;
end Main_Task;

task Back_End is
     -- no longer need an entry to receive data from Main Task
     entry Terminate_Self;
end Back_End;

package Keyboard_Input_Buffer is new
     Buffer (Bufmax => 80, Item => Character);
package Sensor_Input_Buffer is new
     Buffer (Bufmax => 4, Item => <some data type>;
package Output_Buffer is new
     Buffer (Bufmax => 4, Item => Processed_Data_Type);

task body Front_End is

     Oper_Data   : Character;
     Sensor_Data : <some data type>;

begin -- Front_End
     loop
          select
               accept Keyboard_Interrupt
                     (In_Oper_Data : in Character) do
                    Oper_Data := In_Oper_Data;
               end Keyboard_Interrupt;
               Keyboard_Input_Buffer.Put (Data => Oper_Data);
          or
               accept Sensor_Interrupt
                     (In_Sensor_Data : in <some data type>) do
                    Sensor_Data := In_Sensor_Data;
               end Sensor_Interrupt;
               Sensor_Input_Buffer.Put (Data => Sensor_Data);
          end select;

          if <time to terminate> then
               Main_Task.Terminate_Self;
               exit;
          end if;
     end loop;

end Front_End;
```

```
task body Main_Task is

     Out_Data    : Processed_Data_Type;
     Oper_Data   : Character;
     Sensor_Data : <some data type>;

begin
     loop
          Sensor_Input_Buffer.Get (Sensor_Data);
          -- Process the raw sensor data and post it
          -- for Back End processing.
          Out_Data := ...;
          Output_Buffer.Put (Data => Out_Data);
          .
          .
          .
          select
               accept Terminate_Self;
               Back_End.Terminate_Self;
               exit;
          else
               null;
          end select;
     end loop;

end Main_Task;

task body Back_End is

     Output_Data : Processed_Data_Type;

begin --Back_End
     loop
          Output_Buffer.Get (Output_Data);
          --Process the data;
          ...
          select
               accept Terminate_Self;
               exit;
          else
               null;
          end select;
     end loop;
end Back_End;

end Task_Package;
```

Within this package we have made use of three buffers, one for keyboard input, one for sensor input and one for output. Data is passed between tasks via these buffers, not directly via rendezvous with each other. Thus each task will operate at its own pace and will not have to wait until the input data it needs is available. We have incorporated an entry Terminate_Self in each task that is used to terminate processing completely. Thus when Front_End notices that it is time to terminate then Main_Task.Terminate_Self is invoked which in turn invokes Back_End.Terminate_Self. Once all of these tasks are at the end of their bodies then the package body may be exited according to Ada's rules for dependent tasks.

Notice, however, that even when the execution of Front_End, Main_Task and Back_End is complete, the package body may not be exited because of the instantiations of the package Buffer. The three instantiations in the package body give rise to the creation of three more dependent tasks. (Of course we have carefully hidden the fact that we are using a task within the body of Buffer!) These three tasks are not terminated or waiting at an open terminate alternative, and therefore the package body cannot be left according to Ada's rules for task dependency.

Let's solve the problem by providing a mechanism for terminating the task in package Buffer. One solution is to provide a third procedure in the package, say Halt, that is to be called when the buffer is no longer

needed. This procedure could either abort task Buffer_Mgr (an extreme action to take!) or rendezvous with a third entry, Buffer_Mgr.Term, that will terminate the task via an exit of the loop (and an encounter with the task's end statement). While this solution works, it has the drawbacks of introducing another procedure, and thus another interface for a user to worry about (in this case a user must remember to call this procedure when he is done with the buffer). It also introduces the possibility of exceptions being raised if Get or Put is invoked subsequent to a call to Halt because Get or Put will attempt to rendezvous with a terminated task. In Ada such an action will result in the exception Tasking_Error being raised at the point of the entry call.

A second solution without these drawbacks is to add a terminate alternative to the select statement within task Buffer_Mgr:

```
generic

     Bufmax: Natural;
     type Item is private;

package Buffer is

     procedure Put(Data  : in Item);
     procedure Get(Data  : out Item);

end Buffer;

package body Buffer is

     task Buffer_Mgr is
          entry Write(Data  : in Item);
          entry Read (Data  : out Item);
     end Buffer_Mgr;
```

```
        task body Buffer_Mgr is
             Buffer_Area  : array (1 .. Bufmax) of Item;
             Count        : Integer range 0 .. Bufmax := 0;
             Next_Read    : Integer range 1 .. Bufmax := 1;
             Next_Write   : Integer range 1 .. Bufmax := 1;
        begin
             loop
                  select
                       when Count < Bufmax =>
                  -- there is still space in the array.
                            accept Write (Data :in Item) do
                                 Buffer_Area (Next_Write) := Data;
                            end Write;
                            Next_Write := (Next_Write mod Bufmax) + 1;
                            Count := Count + 1;
                  or
                       when Count > 0 =>
                            --there is data to be read.
                            accept Read (Data : out Item) do
                                 Data := Buffer_Area (Next_Read);
                            end Read;
                            Next_Read := (Next_Read mod Bufmax) + 1;
                            Count := Count - 1;
                  or
                       terminate;
                  end select;
             end loop;
        end Buffer_Mgr;

        procedure Put (Data : in Item) is
        begin
             Buffer_Mgr.Write (Data);
        end Put;

        procedure Get (Data : out Item) is
        begin
             Buffer_Mgr.Read (Data);
        end Get;
   end Buffer;
```

According to Ada's rules for task termination, Buffer_Mgr will terminate if it is at an open terminate alternative and if its master, i.e., the innermost task, block statement, subprogram or library package that

created it, is ready to exit or terminate (in the case of a master that is a task). Thus, if the generic package is instantiated in a library package (as in Task_Package) and the package is ready to be left, then the Buffer_Mgr created as a result of the instantiation will be terminated since it is waiting at an open terminate alternative. The use of the terminate alternative in this way should be considered in the design of any task.

Problem

1. The implementation of Buffer developed in this section makes use of a blocking queue. That is, the buffer area acts as a queue in that data is inserted and retrieved in a first-in, first-out (FIFO) manner. Blocking results from the fact that when the queue is empty, a call to Get will result in the caller having to wait until data is Put, and hence is "blocked" from executing further. Similarly, a caller is blocked when there is a Put to a full buffer. Describe the changes to Buffer needed to implement a non-blocking queue. (HINT: Two possibilities are to write over the oldest item in the buffer or to write over the newest item in the buffer. On the Get operation, it is probably acceptable to block the caller).

2. In some systems it may be more useful to allow multiple data items to be retrieved from the buffer at once. Thus, instead of Get returning a single item each time it is called, it will return all of the items in the buffer at the time of the call. (Items are still Put one at a time.) Modify the generic package Buffer to follow this buffering discipline. If there are no items in the buffer when Get is called, then return zero items.

HINT: Change Get to return an array of items and a count of "significant" items being returned in that array. This means that the package will have to declare a type that may be used to declare objects to be passed to Get.

Discussion and Solution

1. In the solution presented here the oldest data in the buffer is replaced by new data whenever the buffer is full and Put is called. Note that the solution is as simple as removing the guard on the accept for Write, i.e "when Count < Bufmax" and guaranteeing that Count can never exceed Bufmax.

The code follows:

```
generic

     Bufmax : Natural;
     type Item is private;

package Buffer is

     procedure Put (Data : in Item);
     procedure Get (Data : out Item);

end Buffer;

package body Buffer is

     task Buffer_Mgr is
          entry Write (Data : in Item);
          entry Read (Data  : out Item);
     end Buffer_Mgr;

     task body Buffer_Mgr is
          Buffer_Area  : array (1 .. Bufmax) of Item;
          Count        : Integer range 0 .. Bufmax := 0;
          Next_Read    : Integer range 1 .. Bufmax := 1;
          Next_Write   : Integer range 1 .. Bufmax := 1;
     begin
          loop
               select
                    -- there is still space in the array.
                    accept Write (Data : in Item) do
                         Buffer_Area (Next_Write) := Data;
                    end Write;
                    Next_Write := (Next_Write mod Bufmax) + 1;
                    if Count < Bufmax then
                         Count := Count + 1;
                    end if;
               or
```

```
                    when Count > 0 =>
                    -- there is data to be read.
                    accept Read (Data : out Item) do
                         Data := Buffer_Area (Next_Read);
                    end Read;
                    Next_Read := (Next_Read mod Bufmax) + 1;
                    Count := Count - 1;
               or
                    terminate;
               end select;
          end loop;
     end Buffer_Mgr;

     procedure Put (Data : in Item) is
     begin
          Buffer_Mgr.Write (Data);
     end Put;

     procedure Get (Data : out Item) is
     begin
          Buffer_Mgr.Read (Data);
     end Get;

end Buffer;
```

Note that another solution is to maintain the select statement as it was originally and add a fourth alternative:

```
          when Count = Bufmax =>
               accept Write (Data :in Item) do
                    Buffer_Area (Next_Write) := Data;
               end Write;
               Next_Write := (Next_Write mod Bufmax) + 1;
```

This solution eliminates the need for an if statement to ensure that Count does not exceed Bufmax. In some sense this second solution is easier to understand than the first one above because it makes the case of a full buffer more apparent, though at the expense of a larger select statement and (very probably) a larger program.

A solution that replaces the data most recently written to the buffer is similar to the ones that replace the oldest data. Care must be taken to ensure that Count does not exceed Bufmax. For purposes of keeping the program understandable it may be more practical to maintain a Last_Write index instead of a Next_Write index into Buffer_Area.

2. In this solution, the type of the parameter to Get is a record type that is defined in the Buffer package specification. The record value returned gives a count of the items retrieved and the list of items, Item_List. The full solution code is presented next.

```
generic

     Bufmax : Natural;
     type Item is private;

package Buffer is

     type Item_List is
          record
               Count : Natural range 0 .. Bufmax;
               Items : array (1 .. Bufmax) of Item;
          end record;

     procedure Put (Data : in Item);

     procedure Get (Data : out Item_List);

end Buffer;

package body Buffer is

     task Buffer_Mgr is
          entry Write (Data : in Item);
          entry Read (Data  : out Item_List);
     end Buffer_Mgr;
```

```
    task body Buffer_Mgr is
        Buffer_Area : Item_List;
    begin
        loop
            select
                when Buffer_Area.Count < Bufmax =>
                --there is still space in the array..
                accept Write (Data : in Item) do
                    Buffer_Area.Count := Buffer_Area.Count + 1;
                    Buffer_Area.Items (Buffer_Area.Count) := Data;
                end Write;
            or
                accept Read (Data : out Item_List) do
                    Data := Buffer_Area;
                end Read;
                Buffer_Area.Count := 0;
            or
                terminate;
            end select;
        end loop;
    end Buffer_Mgr;

    procedure Put (Data : in Item) is
    begin
        Buffer_Mgr.Write (Data);
    end Put;

    procedure Get (Data : out Item_List) is
    begin
        Buffer_Mgr.Read(Data);
    end Get;

end Buffer;
```

A sample use of this new Buffer package is:

```
.
.
.
package My_Buffer is new Buffer (Bufmax =>6, Item => Integer);

In_Data : My_Buffer.Item_List;

     .
     .
     .

          My_Buffer.Put (1);
          My_Buffer.Put (2);
          My_Buffer.Put (3);
          My-Buffer.Get (In_Data);

     .
     .
     .
```

The call to Get will yield a value of

```
My Buffer.Item_List'(Count => 3, Items => (1,2,3, x, y, z))
```

where x, y, and z are unknown values.

Note that in this solution we no longer use Next_Read and Next_Write. In addition, Buffer_Area and Count were replaced by a single variable object, Buffer_Area, of type Item_List. This permits the accept body for Read to be a single assignment statement. Note, however, that in making Buffer_Area of this type and eliminating Next_Write, the accept body for Write has been lengthened, making a rendezvous with Write take longer than before. If this takes "too long", then an alternate solution would be to copy the data passed to Write to a local object of type Item and to update Buffer_Area outside of the accept body, i.e.,

```
X : Item;    -- declared within Buffer_Mgr.
.
.
.
accept Write (Data : in Item) do
     X := Item;
end Write;
Buffer_Area.Count := Buffer_Area.Count + 1;
Buffer_Area.Items(Count) := X;
```

EXERCISE 4.2

MONITORS

Objective

In this section we will introduce the concept of a monitor, a commonly used tool for controlling a system's resources. An Ada implementation of a monitor is developed.

Tutorial

A concept that is commonly used in the design of real-time computer applications is that of a monitor. A monitor is a software module that is responsible for controlling a resource. (Its purpose is to monitor the use of that resource). For example, read and write operations to a disk are usually controlled by a monitor that ensures the integrity of data on the disk.

In Section 3.2 we discussed semaphores and their uses in synchronizing the allocation of resources. The semaphore is an effective low-level synchronization primitive. However, the use of semaphores in a complex application can result in disaster if an occurrence of a semaphore operation is omitted somewhere in the system or if the use of a semaphore is erroneous. A monitor replaces the need to perform operations on semaphores. Entry to a monitor by one process excludes entry by any other process. A monitor thereby ensures that if it has exclusive access to a resource, then a

monitor's user has exclusive access to that resource. (Most computers provide hardware support for monitors by providing a privileged position for them: they can run in uninterruptable mode, they can access special areas of memory, and/or they can execute special instructions such as I/O instructions).

Let's develop a monitor in Ada. Consider a problem of having a pool of data common to a group of processes. The data in the pool may be set by one or more processes or used by one or more processes. Any number of processes should be allowed to read the pool simultaneously, but no reads are to be permitted during a write operation. A monitor will be used to control the reading and writing of data to the pool.

A monitor can be written in Ada as a package whose visible part contains one or more subprogram declarations and potentially some type declarations. Since the data type is not of primary concern, and since it might be useful to use any data type, let's make this pool manager a generic package.

```
generic
     type Data_Type is private;  -- The type of the data to be pooled

     Initial_Value : Data_Type;  -- An initial value for the pool data

package Read_Write_Manager is
     procedure Read (Data  : out Data_Type);
     procedure Write (Data : in Data_Type);
end Read_Write_Manager;
```

The procedure Read is to be used to get the current value of the data pool and the procedure Write is to be used to set the value of the pool. No constraints are imposed upon the order of calls to Read and Write. What is guaranteed, however, is that Read and Write will not interfere with one another. That is, if one process is writing, then another will not be simultaneously reading. Therefore, the integrity of data in the pool is maintained, the entire pool being updated (via Write) before the pool is read or the entire pool being read before the pool is updated.

The body of Read_Write_Manager will contain a declaration of a pool object. Ada guarantees that this object can be accessed only from within the package body. Thus we can be sure that the monitor has complete control over access to the pool. Read and Write are the only means by which the data in the pool can be accessed. Let's consider two possible implementations for the body of Read_Write_Manager, the first using semaphores and the second using the Ada rendezvous mechanism. Recall the essential aspect of our pool manager as being that any number of read operations may occur simultaneously, but only one Write may occur at a time.

Implementation #1

```
with Semaphore;
package body Read_Write_Manager is

     Pool:  Data_Type := Initial_Value;

     Reader_Count     : Natural := 0;
     Write_Wait_Count : Natural := 0;
     Read_Wait_Count  : Natural := 0;

     package Mutex is new Semaphore;
     package Ok_to_Read is new Semaphore;
     package Ok_to_Write is new Semaphore;

     -- Mutex is used as a mutual exclusion semaphore for the operations
     -- on the three counters.  Ok_to_Read and Ok_to_Write are used as signals
     -- to indicate that conditions have changed and either reading or writing
     -- may take place if a writer has completed its update, it will signal
     -- that it is Ok_to_Read.  If no processes are waiting to read, the
     -- writer signals any next writer that it is Ok_to_Write.

     procedure Start_Read is
     begin
          Mutex.Wait;
          if Write_Wait_Count > 0 then
             Read_Wait_Count := Read_Wait_Count + 1;
             Mutex.Signal;
             Ok_to_Read.Wait;
          else
               Reader_Count := Reader_Count + 1;
               Mutex.Signal;
          end if;
     end Start_Read;


     procedure End_Read is
     begin
          Mutex.Wait;
          Reader_Count := Reader_Count - 1;
          if (Reader_Count = 0) and (Write_Wait_Count > 0) then
              Ok_to_Write.Signal;
          end if;
          Mutex.Signal;
     end End_Read;
```

```
        procedure Start_Write is
        begin
             Mutex.Wait;
             Write_Wait_Count := Write_Wait_Count + 1;
             if (Reader_Count > 0) or (Write_Wait_Count > 1) then
                 Mutex.Signal;
                 Ok_to_Write.Wait;
             else
                 Mutex.Signal;
             end if;
        end Start_Write;

        procedure End_Write is
        begin
             Mutex.Wait;
             Write_Wait_Count := Write_Wait_Count - 1;
             if Read_Wait_Count > 0 then
                for I in 1 .. Read_Wait_Count loop
                   Ok_to_Read.Signal;
                   Reader_Count    := Reader_Count + 1;
                   Read_Wait_Count := Read_Wait_Count - 1;
                end loop;
             elsif Wait_Write_Count > 0 then
                Ok_to_Write.Signal;
             end if;
             Mutex.Signal;
        end End_Write;

        procedure Read (Data : out Data_Type) is
        begin
             Start_Read;
             Data := Pool;
             End_Read;
        end Read;

        procedure Write (Data : in Data_Type) is
        begin
             Start_Write;
             Pool := Data;
             End_Write;
        end Write;

    begin     --Read_Write_Manager

        Mutex.Signal;  -- so that the first process can enter the critical
                       -- section protected by mutex.

    end Read_Write_Manager;
```

Implementation #2

```
package body Read_Write_Manager is

     Pool : Data_Type := Initial_Value;

     task Manager is
       entry Start_Read;
       entry End_Read;
       entry Start_Write;
       entry End_Write;
     end Manager;

     procedure Read (Data : out Data_Type) is
     begin
          Manager.Start_Read;
          Data := Pool;
          Manager.End_Read;
     end Start_Read;

     procedure Write (Data : in Data_Type) is
     begin
          Manager.Start_Write;
          Pool := Data;
          Manager.End_Write;
     end Start_Write;

     task body Manager is

          Readers:  Natural := 0;

     begin
          loop
               select
                    when Start_Write'Count = 0 =>
                      accept Start_Read do
                        Readers := Readers + 1;
                      end Start_Read;
                    or
                      accept End_Read do
                        Readers := Readers - 1;
                      end End_Read;
```

```
                    or
                      when Readers = 0 =>
                        accept Start_Write;
                        accept End_Write;
                        for I in 1 .. Start_Read'Count loop
                           accept Start_Read do
                              Readers := Readers + 1;
                           end Start_Read;
                        end loop;
                end select;
           end loop;
      end Manager;

end Read_Write_Manager;
```

This implementation does not use semaphores. The monitor design for the Pool still uses guards but the guards are simple entry calls for synchronization. The attribute Count (of the number of entry calls waiting) is used to control reader access. The design strategy is to provide fairness to Readers who are blocked by Writers by releasing of outstanding Readers after the writer has finished. Also, after a writer requests access to the pool, no more readers are allowed to access the pool until the writer has updated the pool; any reading currently in progress, however, is allowed to complete.

Implementation #3:

```
package body Read_Write_Manager is

     Pool : Data_Type := Initial_Value;

     task Pool_Manager is
          entry Read (Data  : out Data_Type);
          entry Write (Data : in Data_Type);
     end Pool_Manager;
```

```
    task body Pool_Manager is
    begin
        select
            accept Read (Data : out Data_Type) do
                Data := Pool;
            end Read;
        or
            accept Write (Data : in Data_Type) do
                Pool := Data;
            end Write;
        end select;
    end Pool_Manager;

    procedure Read (Data : out Data_Type) is
    begin
        Pool_Manager.Read (Data);
    end Read;

    procedure Write (Data : in Data_Type() is
    begin
        Pool_Manager.Write (Data);
    end Write;

end Read_Write_Manager;
```

This implementation uses the Ada rendezvous mechanism directly to control the data pool. It is simpler than the previous two implementations, but it relies on an Ada implementation's scheduling algorithm. That is, whenever both a Pool_Manager.Read and a Pool_Manager.Write are pending, Ada determines which of the two will rendezvous. In the previous two solutions, the monitor has control over the order of "rendezvous," namely that reads and writes are performed in the order that they are requested.

Problem

1. Modify the third implementation approach so that after a writer requests access, no reader is allowed to access the pool, and after the writer completes, all pending readers are accepted before another write is accepted. Just write the code for the Pool_Manager body.

2. What would be the effect in implementation 2 if the guard, when Readers = 0 were replaced with when Start_Read'Count = 0?

Discussion and Solution

1. The same approach is used as in Implementation #2 in the Tutorial. The code follows:

```
task body Pool_Manager is
     Readers : Natural := 0;
begin
     select
          when Write'Count = 0 =>
               accept Read (Data : out Data_Type) do
                 Data := Pool;
               end Read;
     or
          when Readers = 0 =>
               accept Write (Data :in Data_Type) do
                 Pool := Data;
               end Write;
               for I in 1 .. Read'Count loop
                    accept Read (Data : out Data_Type) do
                           Data := Pool;
                    end Read;
               end loop;
     end select;

end Pool_Manager;
```

2. Consider the effect if both a reader and a writer request access to the pool before the Manager has started executing the select statement. Then Start_Write'Count = 1 and Start_Read'Count = 1. The only open alternative is the one for End_Read, which can never be called. In short, the system deadlocks.

CHAPTER 5

SPECIAL-PURPOSE TASKING FEATURES

5.1 Advanced Tasking Concepts

EXERCISE 5.1

ADVANCED TASKING CONCEPTS

Objective

In this section we will discuss features of Ada that are useful in solving problems associated with real-time applications. We will look in detail at task attributes, the abort statement, exceptions raised during task communication, task priorities, entry families, shared variables and interrupts.

Tutorial

There are six attributes associated with tasks and entries:

Address,
Callable,
Count,
Size,
Storage_Size and
Terminated.

Given a task type T, a task object X and a task entry E, the attributes are defined as follows.

T'Address is the address of the first machine code instruction associated with the body of task type T. The type of this value is System.Address.

E'Address is the address of the first machine code instruction associated with the entry E. (If an address clause has been given for E, then the value of this attribute corresponds to a hardware interrupt). The type of this value is System.Address.

X'Callable yields a Boolean value of False when X is either completed or terminated, or when X is abnormal. (Recall that X is completed when it has reached the end of the statements in its body; X is terminated when X is completed and any dependent tasks are either completed or at an open terminate alternative in a select statement; and X is abnormal when X has been named in an executed abort statement). Otherwise this attribute yields the value False.

E'Count yields the number of entry calls presently queued on the entry E. (If the attribute is evaluated by the execution of an accept statement for E the count does not include the calling task). The value of this attribute is of type universal integer.

T'Size yields a universal integer value that is the number of bits allocated to hold a task object of type T. This bit count corresponds to the size of the code associated with the task object.

X'Size yields a universal-integer value that is the same as T'Size where T is the type of X.

T'Storage_Size yields a universal integer value that is the number of storage units reserved for each activation of a task of the type T. The value corresponds to the size of the data space required, not the code space required.

X'Storage_Size yields a universal integer value that is the same as T'Storage_Size where T is the type of X.

X'Terminated yields the boolean value True if the task designated by X is terminated and False otherwise.

These attributes have applications in some real-time environments. One of the most useful ones is 'Count. 'Count is used in an implementation of the data pool in Section 4.2 to determine how many Start_Reads should be done before another write operation is enabled. This use illustrates a situation in which two entries in a task synchronize the order in which rendezvous happen: one entry (End_Write) ensures that all calls to another entry (Start_Read) are "flushed" before processing is continued. It is possible to do this without using the 'Count attribute, but contrast these two solutions:

Solution 1:

```
for I in 1 .. Start_Read'Count loop
     accept Start_Read do
          Readers := Readers + 1;
     end Start_Read;
end loop;
```

Solution 2:

```
loop
     select
          accept Start_Read do
               Readers := Readers + 1;
          end Start_Read;
     else
          exit;
     end select;
end loop;
```

The second solution loops until no more rendezvous with Start_Read are possible. The solutions, however, are not equivalent because in the first solution the number of iterations is determined prior to loop entry whereas in the second solution the number of iterations is determined dynamically. If some task calls Start_Read once the loop has started in Solution 1, it will not rendezvous until sometime after the end of the for loop. However, if some task calls Start_Read once the loop has started in Solution 2, then it may rendezvous before the end of the loop. Thus 'Count gives us a little more control.

The number of queued entries can decrease if one of the calling tasks is aborted or if the calling task has issued a timed entry call and the timeout delay has expired. If the number of tasks actually queued is less than the number of accept statements to be executed within the loop in the first solution, then execution will eventually stop at the accept statement, waiting for a call. To avoid this possibility, the accept statements should be written so they do not wait if there is no queued entry call.

```
select
     accept Start_Read do
          Readers := Readers + 1;
     end Start_Read;
else      -- do nothing if there is no call;
     null;
end select;
```

'Count may be used to prioritize rendezvous within a task. Consider a task T with these entries:

```
Task T is
     entry A;
     entry B;
     entry C;
end T;
```

If it is to be the case that any calls to A should be handled before those to B, and calls to B should be handled before those to C, then an implementation of the task body can make use of 'Count.

```
task body T is
     A_Count : Natural;
     B_Count : Natural;
begin
     loop
          A_Count := T.A'Count;
          B_Count := T.B'Count;
          select
               when A_Count > 0 =>
                    accept A;
          or
               when (A_Count = 0) and (B_Count > 0) =>
                    accept B;
          or
               when (A_Count = 0) and (B_Count = 0) =>
                    accept C;
          else
               null;
          end select;
     end loop;
end T;
```

The select statement is used to set up the rendezvous and guards involving conditions on the count for each of the three entries are used to control

the valid rendezvous on each iteration of the loop. Note that if there are no entry calls to any of A, B, or C, then the task keeps looping over a null statement. If calls to A, B, and C are infrequent, then it may be best to write guards that leave all attention open when no entry calls are waiting.

The attributes 'Callable and 'Terminated provide information about the state of a task. These attributes are sometimes useful if an entry to be called is potentially unavailable because the task that it is in has finished its execution. An attempt to rendezvous with such an entry will result in a Tasking_Error exception being raised. These attributes provide a means of avoiding the exception. Code of the form

```
if T'Callable then
     T.A;
end if;
```

is easier to understand than

```
begin
     T.A;
exception
     when Tasking_Error =>
          null;
end;
```

Note, however, that even if T'Callable is true at the time it is evaluated, T.A may raise Tasking_Error, since T may become complete after T'Callable is evaluated.

These attributes can also be used to provide information about resource status. For example, if a task is assigned to control a device, then if the device is unavailable the task can be made unavailable. In this way the task is the device to other system components using it. If the task is 'Terminated or not 'Callable, then the device is unavailable. Any need for a flag to give the status of the device (up or down) is not necessary because it is implied by the callability of the task.

The Abort Statement

A task normally terminates by its execution reaching the end of the statements in its body or by the execution of a terminate alternative in a select statement, and by the normal termination of its dependent tasks. By this mechanism a task has control over its own demise. However, Ada provides another mechanism for task termination, the abort statement. The statement permits a task to be terminated from anywhere in the program that the task is visible. The format of the abort statement is:

```
abort name {, name};
```

where the names are task names. So if Radar is a task we could abort it by executing the statement:

```
abort Radar;
```

The effect of this statement is to place the named task(s) into an abnormal state. (If more than one task is named then the order that they are aborted is undefined; if order is important use separate abort

statements). An attempt to abort a task that has already completed has no effect. When a task is in an abnormal state then any calls to an entry in that task result in exception Tasking_Error being raised. If a rendezvous is in progress when the abort is executed, however, it is allowed to finish; Tasking_Error is then raised in the calling task.

An abnormal task, i.e., one that is named in an executed abort statement, completes its execution no later than when it reaches one of the following: the end of its activation; a point where it causes the activation of another task; an entry call; the start or end of an accept statement; a select statement; a delay statement; an exception handler; or an abort statement. It is important to remember that a task does not necessarily have to complete before an abort statement naming it finishes executing. It is also important to note that if abnormal completion of a task takes place while the task is updating a variable, then the value of that variable is undefined. Let's look at an example.

```
package P is

     X : Natural := 0;    -- Counts times through loop in T
     Y : Natural := 0:    -- Counts calls to T.E
     pragma shared (X);   -- the need for these pragmas will be
     pragma shared (Y);   -- discussed later.
     task T is
          entry E;
     end T;
end P;
```

```
package body P is

     task body T is
     begin
          loop
               X := X + 1;
               select
                    accept E do
                         Y := Y + 1;
                    end E;
               else
                    null;
               end select;
          end loop
     end T;
end P;
```

Consider a segment of code that uses P:

```
loop
     if P.X > 1000 then
          Count := P.Y;
          abort P.T;
          exit;
     else
          P.T.E;
     end if;
end loop;
P.T.E;
```

This loop makes entry calls to P.T.E. as long as P.X is not greater than one-thousand. Once P.X reaches one-thousand the task is aborted and the loop is left. When the entry call, P.T.E, following the end of the loop is executed, exception Tasking_Error will be raised. The value of Y will be the number of times that P.T.E successfully rendezvoused with callers. It is not necessarily true that after the loop, Count = P.Y, because a call to P.T.E from another task may occur between the assignment of P.Y to Count and the end of the abort statement's execution. (Recall that P.T becomes abnormal but any rendezvous in progress will complete.)

The abort statement is an extreme measure to take to terminate a task. If a running task must be terminated, consider including an entry that will perform orderly task completion from within itself. This mechanism permits the termination process to be better understood by the task's users.

Task Priorities

Ada provides a mechanism for associating a task with a priority. With this mechanism a task having a higher priority is considered to be more urgent and its processing is scheduled before those tasks having lower priority. A task's priority is established by a pragma that appears in its specification:

```
pragma Priority (expression);
```

The expression must be of type Priority, a subtype of Standard.Integer that is defined in package System. The expression must be static. A higher value indicates a higher priority. Thus a task having priority 3 has a higher degree of urgency than one having priority 1. Task priorities are established at compilation time and cannot be changed dynamically. (Note that the valid range of values for priority is implementation-defined.) If a task is not given an explicit priority by this pragma, then the compiler will assign a priority.

Priority does not affect the order in which entry calls are served. Ada defines this order as FIFO. Priority is used to assist in the

allocation of processor resources. If two tasks are both eligible for execution then the one having the higher priority is selected for running. If these two tasks have equal priorities, then the one selected is determined by the runtime environment. In the case of equal priorities, the scheduling algorithm need not even be fair. Consider some examples using the following task specifications:

```
task High is
     pragma Priority (3);
     entry E;
end High;

task Medium_1 is
     pragma Priority (2);
     entry E;
end Medium_1;

task Medium_2 is
     pragma Priority (2);
     entry E;
end Medium_2;

task Low is
     pragma Priority (1);
     entry E;
end Low;
```

Tasks eligible for execution on the same processor	Task selected for execution
High, Medium_1, Medium_2, Low	High
Medium_1, Medium_2, Low	either Medium_1 or Medium_2
Medium_1, Low	Medium_1

If two tasks are in rendezvous, then the rendezvous is executed with the higher priority. Note that this is the case even when the task having the lower priority is the one receiving the entry call. Thus if the body

of High makes a call to Low.E, then the rendezvous is executed with priority three even though Low has priority one.

Since the semantics of priorities are only specified in Ada by guidelines, it is not a good idea to use priorities to achieve task synchronization. Different implementations of Ada, or even revision to a runtime environment could adversely affect a system designed in that way. Remember that priorities should be used only to show urgency.

Exception raised during task communications.

Just as a subprogram call can result in the raising of an exception, so can an entry call. For example, if an actual in parameter does not satisfy the constraints of the formal, then Constraint_Error is raised before the entry call. An exception raised during the evaluation of an actual parameter results in the call being ignored and the transfer of execution to the appropriate handler for that exception. However, there are some other cases in which an exception can arise at an entry call.

The predefined exception Tasking_Error is raised at the point of the call if the called entry is in a task that has already completed or is completed before it accepts the entry call. (In this case, there is no rendezvous with the called entry.)

If the task containing a called entry is aborted during a rendezvous, then the exception Tasking_Error is raised at the point of the call. On the other hand, if the task issuing the entry call is aborted, no exception is raised and the rendezvous is completed before the task becomes abnormal.

The raising of an exception within an accept statement has an interesting effect. If there is an exception handler within an inner frame, then that handler is executed and the exception is propagated no further. For example, the accept body

```
accept E (B : out Boolean)
     begin
         raise Constraint_Error;
     exception
         when Constraint_Error =>
            B := False;
     end;
end E;
```

will always result in Constraint_Error being raised, but because there is a handler for that exception there is no propagation of the exception beyond the inner frame (i.e., the begin-exception-end statement). However, if there had been no handler, then the exception would be propagated to <u>two</u> places - at the point of the entry call that is in rendezvous with the accept, and to a handler of the frame enclosing the accept statement. For example, given the segments

```
task body T1 is
     .
     .
     .
     T2.E (B);
     .
     .
     .
end T1;
```

```
task body T2 is
     .
     .
     .
     accept E (B: out Boolean)do
       raise Constraint_Error;
     end E;
     .
     .
     .
end T2;
```

then the rendezvous between T1 and T2 at E will result in Constraint_Error being raised at both the point of the entry call in T1 and at the statement following "end E;" in T2. Thus the exception is propagated in two directions.

Exceptions raised within a rendezvous are sometimes useful. Consider the buffering package developed in Section 4.1, for example. In that package, a call to entry Write will result in the caller's waiting for available space when the buffer is full. A way to change this is to have exceptions raised in the case of a Write to a full buffer and a Read of an empty buffer. We will want to propagate these exceptions out of Get and Put as well. The package specification then becomes:

```
generic
     Bufmax : Natural;
     type Item is private;
package Buffer is
     Buffer_Empty : exception;     -- Raised if the buffer
                                   -- is empty when let is
                                   -- called.
     Buffer_Full  : exception;     -- Raised if the buffer is
                                   -- full when Put is called.
     procedure Get (Data : out Item);
     procedure Put (Data : in  Item);
end Buffer;
```

This specification provides more flexibility in that the caller can determine what to do if the buffer is full or empty. The disadvantage is that the caller must perform a busy wait to get the behavior of the package specified without exceptions. So, for example, if we want to wait for the next data item even when the buffer is empty, we have to loop:

```
loop
     begin
         Get (Data);
         exit;
     exception
         when Buffer_Empty =>
           delay 0.1;
     end;
end loop;
```

If we want to handle this situation in the most efficient way, one solution is to have four procedures in the package: Get_No_Wait, Get_Wait, Put_No_Wait and Put_Wait. The interface is larger, but the user correspondingly has more options.

Entry families

Up to this point we have considered "single entries," i.e., an entry that is not part of an "entry family." An entry family is a collection of entries, each entry of which is identified by an index value similar to the way that an array component is identified by an index value. An entry family is declared by specifying a discrete range within parentheses immediately following the entry name. For example,

```
task T is
     entry Fam (Boolean) (X : in Data_Type);
     entry E;
entry T;
```

declares a task having three entries, Fam(False), Fam(True) and E. A call to an entry that is a member of a family includes a value in parentheses that designates the entry, for example:

```
T.Fam (True)(X => Some_Value);
```

or

```
T.Fam (X = Y or X > Z) (X => Some_Value);
```

The accept statement for an entry family takes a value in parenthesis to designate which one of the family is being designated, for example:

```
accept Fam (True)(X : in Data_Type)
```

or

```
accept Fam (B)(X : in Data_Type)  -- B is a Boolean object
```

An entry family is useful when there is a group of entries that have the same processing or when instead of writing an accept for each entry, one accept is sufficient. Consider an application in which we wish to write a task to route messages. Messages have three classes: A, B and C. Messages should be processed in a round-robin fashion, i.e., in the order A, then B, then C, then A, again and so on. Assume that each message is placed in a buffer as it is received. The following declarations are visible:

```
-- use Buffer package declared in Section 4.1
with Buffer;   -- from Section 4.1
type Message is String (1 .. 80);
type Class is (A, B, C);

package Message_Buffer is new Buffer
                              (Bufmax => 10, Item => Message);
```

We define the routing task as

```
task Route_Message is
     entry Send (Class) (M : in Message);
end Route_Message;

task body Route_Message is
begin
     loop
          for I in Class loop
               accept Send (I) (M : in Message) do
                    Message_Buffer.Put (M);
               end Send;
          end loop;
     end loop;
end Route_Message;
```

The accepted calls are in the order Send (A), Send (B), Send (C), Send (A) and so on. In this way, we have generated a round robin discipline for sending messages with a single accept statement. The solution is also reasonable in terms of the users of Send. The class of the message must be specified. To make it an index to the entry family instead of a parameter to the call gives the class a certain distinction. And making the class an index instead of using three separate entries (e.g., Send_A, Send_B, and Send_C) permits a caller to designate the class using an expression. The call

```
Route_Message.Send(C)(Msg)
```

is more convenient than

```
case C is
     when A =>
         Route_Message.Send_A(Msg);
     when B =>
         Route_Message.Send_B(Msg);
     when C =>
         Route_Message.Send_C(Msg);
end case;
```

Shared Variable

From a task's point of view, it has exclusive access to variables that are nonlocal. Thus, for example, if a task uses a variable that is declared in a package specification, then the task has exclusive use of that variable. From the task's viewpoint, the only time that variable is updated is if the task updates it. One implication of this is that a compiler may optimize the use of the variable, keeping its value in a register, for example, instead of retrieving the value from memory each time. A problem can arise, therefore, if two tasks use the same nonlocal variable. The value of the variable will change in an unpredictable manner depending on how the tasks are scheduled and how a compiler optimizes the references to the variable.

Ada provides a mechanism to share variables among two or more tasks, the Shared pragma. This pragma must follow the declaration of the object to be shared. Furthermore, a shared object must be of a scalar or an access type. (The compiler will issue a warning otherwise). For example,

```
X:  Integer := 0;
pragma Shared (X);
```

declares a variable X and designates it as being shared between two or more tasks. Note that Shared is not intended for use in synchronizing tasks. If two tasks simultaneously update a shared variable, the results are not predictable.

In general, it is not a good practice to use shared variables for task communications. The rendezvous mechanism is the preferred method. However, some situations require the use of shared variables. For example, some computers use memory-mapped I/O to interface with certain devices. Consider a computer that interfaces with a keyboard using memory location 100 (hexadecimal). When a key is struck, the value of the character is written to storage location 100 and an interrupt occurs. We want to have a task that provides the data coming from the keyboard. A solution is:

```
task Keyboard is
     entry Get (C: out Character);
     entry Strike;
     for Strike use at 16#F100#; -- vector address for keyboard
                                        interrupt
end Keyboard;

task body Keyboard is
     Loc_100 : Character;
     pragma Shared (Loc_100);
     for Loc_100 use at 16#100#;
begin
     loop
```

```
            accept Strike;
            accept Get (C :out Character)
                  C := Loc_100;
            end Get;
         end loop;
      end Keyboard;
```

The hardware that places a character in location 100 and generates a keyboard interrupt is treated as a task that makes an entry call to Keyboard.Strike and passes data through memory location Loc_100. Thus, Loc_100 is a variable shared between the hardware task and Keyboard. (Note that this task is implicitly defined within the body of task Keyboard since that is where we declare Loc_100). The entry Get is provided to retrieve this character. The body of the task is fairly simple-minded, requiring that a call to Keyboard.Get be made before accepting another keyboard strike. Depending on system requirements and on the way that an Ada implementation treats interrupts as entry calls (i.e. as an ordinary entry call, a timed entry call, or a conditional entry call), the body may need a more sophisticated algorithm.

Interrupts

Most high-level languages do not directly support hardware or software interrupts. An interrupt handler is usually an assembly language subprogram or a subprogram that is written in a high-level language that is

established as an interrupt handler via a call to an operating system service routine. Ada treats interrupts as entry calls to tasks. Assuming that an interrupt results in a jump to a certain address on the underlying computer an Ada task can be attached to the interrupt via an address clause:

```
task Interrupt is
     entry Reset;
     for Reset use at 16#FFFF#;
end Interrupt;

task body Interrupt is
begin
     loop
         accept Reset;
         -- do something
     end loop;
end Interrupt;
```

We assume here that the reset interrupt vectors (jumps) to hexadecimal address FFFF. When a Reset interrupt occurs, it is treated as an entry call to Interrupt.Reset. It is as though there is a hardware task issuing the call. The priority of such a (imaginary) hardware task is higher than that of any other task. The entry call may be treated by the implementation as an ordinary entry call, a timed entry call or a conditional entry call. If data is associated with an interrupt then this data may be passed to an entry associated with this interrupt as one or more in parameters.

Note that in the above example there is nothing to keep Interrupt.Reset from being called directly (as long as it is visible). A call will have the same effect as one resulting from a Reset interrupt except for any side-effects in the hardware that result from an actual hardware interrupt.

Note that all interrupts on your target computer may not be accessible to your Ada program. For example, if the computer has an interrupt for arithmetic overflow, then more than likely the compiler is reserving the handling of this interrupt to itself for the purposes of recognizing the Numeric_Error exception. It may be tempting to use a timer interrupt in certain time-critical, real-time applications. However, this timer interrupt may be used by the compiler to schedule tasks and to implement delay statements. Therefore, always be sure that an interrupt is accessible before including it as a part of your system design.

Problems

1. Implement the body for package Buffer whose specification appears in the section concerning exceptions and task communications.

2. Assume that there are five tasks that must be run over a period of eight seconds as follows:

SECOND:	0	1	2	3	4	5	6	7
TASK:	T_0	T_1	T_0	T_2	T_0	T_3	T_0	T_4

Each task is responsible for its own cyclic execution via a call to an entry, Scheduler.Startup. The information needed by Startup is the second in the cycle at which it is to be activated. Specify the task Scheduler and implement it.

3. Implement the scheduler in Problem 2 but assume that there is a timer available that interrupts the processor and vectors to location 16#100#. The timer interrupt also provides an integer value of 0 through 7 that is the number of the second in the cycle.

Discussion and Solution

1. The addition of exceptions to the interface requires only a small change to the package body for Buffer as it appears in Section 4.1. Two alternatives have been added to the select statement to recognize the exception cases. The essential part of the change is that when one of the exceptions is raised in the accept body a handler must be available within that body to keep it from propagating any further. If it were to propagate to the end of the task, the task would complete and the buffer would no longer be available. In that case a Tasking_Error exception would propagate out of every call to Put or Get. The solution is as follows:

```
generic

     Bufmax : Natural;
     type Item is private;

package Buffer is

     Buffer_Full  : exception;
     Buffer_Empty : exception;
     procedure Put (Data : in Item);
     procedure Get (Data : out Item);

end Buffer;

package body Buffer is

     task Buffer_Mgr is
         entry Write (Data : in Item);
         entry Read (Data  : out Item);
     end Buffer_Mgr;
```

```
task body Buffer_Mgr is
    Buffer_Area  :  array (1 .. Bufmax) of Item;
    Count        :  Integer range 0 .. Bufmax := 0;
    Next_Read    :  Integer range 1 .. Bufmax := 1;
    Next_Write   :  Integer range 1 .. Bufmax := 1;

begin
    loop
          select
               when Count < Bufmax =>
               --there is still space in the array.
                    accept Write (Data : in Item) do
                        Buffer_Area (Next_Write) := Data;
                    end Write;
                    Next_Write := (Next_Write mod Bufmax)+1;
                    Count : = Count + 1;
          or
               when Count > 0 =>
                    --there is data to be read.
                    accept Read (Data : out Item) do
                         Data := Buffer_Area (Next_Read);
                    end Read;
                    Next_Read := (Next_Read mod Bufmax) + 1;
                    Count := Count - 1
          or
               when Count = Bufmax =>
                    --the buffer is full.
                    accept Write (Data : in Item) do
                         begin
                             raise Buffer_Full;
                         exception
                             when Buffer_Full =>
                                  null;
                         end;
                    end Write;
          or
               when Count = 0 =>
                    --the buffer is empty.
                    accept Read (Data : out Item) do
                         begin
                             raise Buffer_Empty;
                         exception
                             when Buffer_Empty =>
                                  null;
                         end;
                    end Read;
          or
```

```
                terminate;
            end select;
        end loop;
    end Buffer_Mgr;

    procedure Put (Data : in Item) is
    begin
        Buffer_Mgr.Write (Data);
    end Put;

    procedure Get(Data : out Item) is
    begin
        Buffer_Mgr.Read (Data);
    end Get;

end Buffer;
```

An alternative solution is to remove the guards from the alternatives in the select statement and check for the exception cases within the accept bodies. The advantage of this solution is that there are only two accept statements, probably resulting in less space and runtime overhead over the solution proposed above. However, this would be at the cost of increased time spent in rendezvous since in the alternative solution the update of Count, Next_Write and Next_Read would need to be done in the accept body.

2. The information needed by Startup, the number of the second, can be handled either as an entry parameter or as an entry family index. If it is passed as an entry parameter, then the rendezvous would have to last until the tick corresponding to the specified second has arrived. Other entry calls, possibly those requesting at a tick that arrives before the most recent one requested will be blocked. A missed cycle would result. On the other hand, if the number of the second is passed as an entry family index,

then the semantics of Ada tasking will do exactly what we need. The request will be queued until we issue an accept for that particular member of the family. Consider the following solution. (The task is enclosed in a package so that the type Tick can be defined.)

```
package Solution is

     type Tick is Integer range 0 .. 7;

     task Scheduler is
         entry Startup (Tick);
     end Scheduler;

end Solution;

package body Solution is

     task body Scheduler is
     begin
         loop                                  -- cycle continuously over
               for I in Tick loop              -- tick range
                    accept Startup (I);        -- accept startup request
                    delay 1.0;                 -- for current second
               end loop;                       -- delay 1 second till
         end loop;                             -- next rendezvous
     end Scheduler;

end Solution;
```

The algorithm consists of continously cycling over the values zero through seven, the range of Tick, accepting a call to Startup for the current tick number and delaying one second until the next rendezvous. The solution is simple, but there are at least three problems:

1. If no task has scheduled itself for some value of I, then Scheduler will wait until some task does. While this situation doesn't occur in the problem as stated, it is probably a good idea to anticipate the problem.

2. Similarly, if two tasks request to be scheduled at the same time, only one will be started up during a given cycle.

3. The delay of 1.0 seconds is probably going to have things running behind schedule since it is improbable that the rest of the loop takes no time. Besides, Ada only guarantees at <u>least</u> a delay of 1.0 seconds, not an exact delay.

Given these problems, consider another implementation of the body of Scheduler.

```
task body Scheduler is
begin
     loop                                   -- cycle continuously over
         for I in Tick loop                 -- tick range
               for J in 1 .. Scheduler.Startup (I)'Count loop
                    -- accept all tasks scheduled for Startup
                    -- (issuing calls to Startup) on this second.
                    accept Startup(I);
               end loop;
               Clock.Synchronize;           -- marks off one second ticks
         end loop;                          -- of clock, a task to be defined
     end loop;                              -- in package body of Solution.
end Scheduler;
```

This solution handles the first and second problems described above by issuing as many accepts for an entry as there are calls queued for it at any particular time. This solution is particularly nice because it avoids

a problem with a task rescheduling itself too quickly. Recall that E'Count gives the number of calls queued for entry E at the time the attribute is requested. If there are N queued, then N accepts will be issued. Thus, if a task schedules itself very quickly and thereby gets a call back in the queue for Startup (I), it won't be handled until the next full cycle. If, on the other hand we had used a while loop, e.g.,

```
while Startup (I)'Count > 0 loop
    select
         accept Startup (I);
     else
         null;      -- do nothing if there is no call
    end select;
end loop;
```

then we could loop forever in a task that schedules itself quickly, e.g.,

```
loop
    Scheduler.Startup (0);
    Time := Time + 1;
end loop;
```

The call to Clock.Synchronize is an entry call that demarcates one second ticks of the system clock. It is to be defined within the package body of Solution and uses the predefined package Calendar.

```
task Clock is
     entry Synchronize;
end Clock;
```

```
task body Clock is
     Interval  : Calendar.Duration := 1.0;   --1 second intervals
     Next_Time : Calendar.Time     := Calendar.Clock + Interval;
begin
     loop
         delay Next_Time - Calendar.Clock;
         accept Synchronize;
         Next_Time := Next_Time + Interval;
     end loop;
end Clock;
```

This solution still does not guarantee that we will hit second ticks exactly, but it does use a closer value than 1.0 on each iteration of the loop.

3. The availability of a timer interrupt solves some of the difficulties of the above solutions.

```
package Solution is

     type Tick is Integer range 0 .. 7;

     task Scheduler is
         entry Startup (Tick);
     end Scheduler;

end Solution;

package body Solution is

     task Clock is
         entry Synchronize (T : out Tick);
         entry Interrupt (T : in Tick);
         for Interrupt use at 16#100#;
     end Clock;
```

```
        task body Clock is
            N : Tick;
        begin
            loop
                  accept Interrupt (T : in Tick) do
                       N := T;
                  end Interrupt;
                  accept Synchronize (T : out Tick) do
                       T := N;
                  end Synchronize;
            end loop;
        end Clock;


        task body Scheduler is
            T : Tick;
        begin
            loop
                  Clock.Synchronize (T);
                  for J in 1 .. Scheduler.Startup(T)'Count loop
                       select
                            accept Startup (T);
                       else
                            null;
                       end select;
                  end loop;
            end loop;
        end Scheduler;

    end Solution;
```

In this solution we use the tick number returned by the timer. The availability of this interrupt guarantees us a rendezvous every second with Clock.Interrupt. Assuming that processing times is short in Scheduler, then there is no worry about the timer interrupts being lost or queued. (The exact behavior of an entry call to Clock.Interrupt is system-dependent). As long as Clock is always at the "accept Interrupt" statement when the interrupt occurs the behavior has no significant effect on our scheduler.

CHAPTER 6

SCHEDULING AND OPTIMIZATION

6.1 Scheduling and Optimization

EXERCISE 6.1

SCHEDULING AND OPTIMIZATION

Objective

In this section we will look at considerations to make in designing Ada tasks. These considerations concern the selection of a tasking structure (i.e., the identification of Ada tasks and their pattern of interaction). We will consider several examples that represent the types of problems encountered in the design of real-time applications.

Tutorial

SCHEDULING: ADAPTIVE SCHEDULING

In many real-time systems processing must adapt to changes in the operating environment. For example, the algorithms used for navigational processing onboard an aircraft depends on which sensors are operating reliably. Consequently there is a need to schedule the execution of tasks explicitly. A scheduling algorithm is not specified by Ada. The only way to schedule tasks explicitly is to use a scheduler.

One way to implement scheduling is for a task to "check in" at some point in its processing for a signal to continue. The scheduler is aware of the environment and returns a 'yes' or a 'no' in response to a question of "Continue?".

```
package Scheduler is

     type Task_Name is (Navigation, Master_Display, ... );
     function Continue (T : in Task_Name) return Boolean;
end Scheduler;

package body Scheduler is
     ...
     function Continue (T : in Task_Name) return Boolean is
     begin
          ...
          return (Should_T_Be_Running);
     end Continue;
begin   -- Scheduler
     ...
end Scheduler;

task Nav;

task body Nav is
begin
     loop
          while not Scheduler.Continue(Navigation) loop
               delay 0.001;
          end loop;
          Process;
     end loop;
end Nav;
```

The processing details are not shown. The navigation task, Nav, waits until it is to run and then processes some data. The delay within the wait loop is to avoid a busy wait. Thus the navigation task can be "awakened" to process data at certain times determined by the delay. (Recall that the delay is for at least 0.001 seconds). The scheduler is implemented as a package. Perhaps it has an array that defines the current configuration of tasks that are to be running. The array is set via information being received from sensors and perhaps even other tasks.

This solution is easy to understand but wasteful of resources. The navigation task and each other task to be scheduled are constantly executing, using valuable processing resources. In addition, the scheduler cannot control the exact time that a task is awakened. It is up to a task to time its check-in.

Besides check-in, another way to control scheduling is to use a rendezvous. A task awaiting rendezvous is suspended until rendezvous is possible. If a task must rendezvous with the scheduler, then from the scheduler's point of view the task's being blocked is equivalent to the task's being suspended. When the scheduler activates the rendezvous, it awakens the task. There are many ways to effect scheduling using rendezvous.

One way for the scheduler to control the application tasks is a way similar to check-in, but instead of asking the question "Continue?" the task directs "Tell me when to do some more."

```
task Scheduler is
     entry Navigation;
     entry Master_Display;
     .
     .
     .
end Scheduler;
```

```
task body Scheduler is
     ...
begin
     loop
          case Configuration is
               when Full_Operation =>
                    select
                         accept Navigation;
                    or
                         accept Master_Display;
                         ...
                    end select;
               when Attach_Mode =>
                    select
                         accept Master_Display;
                    or
                         ...
                    end select;
               ...
          end case;
     end loop;
end Scheduler;

task body Nav is
begin
     loop
          Scheduler.Navigation;  -- check in
          Process;
     end loop;
end Nav;
```

In this implementation the scheduler determines those tasks that are to execute under the current configuration. The selective wait statement allows the appropriate tasks to continue. For example, the Nav task will wait when the mode changes to Attach_Mode.

A third approach to scheduling is to explicitly create and terminate tasks according to the configuration. The scheduler determines the configuration, creates the appropriate tasks, and waits for the configuration to change. At that point, tasks to be

terminated are aborted and new tasks created. This technique is applicable only where the tasks to be scheduled can be freely stopped and restarted. All tasks must be designed to ensure that their termination via an abort statement does not have disastrous side-effects.

A fourth approach is similar to the third, but avoids the creation and termination of tasks by giving the scheduler the ability to start and stop a task via entry calls.

```
task Nav is
     entry Start;
     entry Stop;
end Nav;

task body Nav is
begin
     loop
          accept Start;
          loop
               select
                    accept Stop;
                    exit;
               else
                    Process;
               end select;
          end loop;
     end loop;
end Nav;
```

The Nav task will execute continuously once a call to Start has been received until a call to Stop is received. The task may be scheduled by alternating calls to Start and Stop. Note that with this implementation the scheduler should have a higher priority than the

tasks that it schedules in order that it can respond quickly to changes in the environment.

SCHEDULING: CYCLIC SCHEDULING

Another form of scheduling that occurs in real-time applications is cyclic scheduling. The execution of a task is synchronized with the ticks of a clock over a specific period of time at which point the execution pattern repeats. The period of time is frequently termed a "major frame" and each interval of time between consecutive ticks is termed a "minor frame." The exercises for Section 5.1 addressed cyclic scheduling using entry families. The family index is the number of the minor frame within a major frame.

OPTIMIZATION

Communication between Ada tasks is asymmetrical. An important design decision to be made with respect to system performance is determining the direction of entry calls.

Consider a problem in which there are two producer tasks and a consumer task. The consumer task is to process the data produced by the other tasks, but the data can arrive in any order. The data must be consumed as soon as it is available.

```
task Producer_1 is
     entry Get (D : out Data_Type);
end Producer_1;
```

```
task Producer_2 is
     entry Get (D : out Data_Type);
end Producer_2;
```

The body of the consumer uses a flag to indicate that data has been received:

```
...
Got_It := False;
while not Got_It loop
     select
          Producer_1.Get (D => x);
          Got_It := True;
     else
          select
               Producer_2.Get (D => x);
               Got_It := True;
          else
               null;
          end select;
     end select;
end loop;
-- Got it now - in x!
```

The conditional entry calls within the loop are necessary to ensure that we get the first data produced. We are polling the two producers until one of them provides data, we are in a busy wait. The loop executes continuously until a rendezvous finally becomes possible. As such, valuable processor resources are being wasted.

Part of the problem is that Ada does not provide a statement for entry calls that is analogous to a selective wait. It might be nice to write

```
select
     Producer_1.Get (D => x);
or
     Producer_2.Get (D => x);
end select;
```

but this is not possible. We could add a delay within the loop to free the processor for some time each iteration, but that would not satisfy the requirement that we process the data as soon as it is available.

The way to avoid the busy wait in this example is to reverse the direction of the rendezvous. If the producers call an entry in the consumer, then the consumer will be suspended until data is available.

```
task Producer_1;

task Producer_2;

task Consumer is
     entry Process (D : in Data_Type);
end Consumer;

task body Producer_1 is
begin
     ... -- get data
     Consumer.Process (Data);
     ...
end Producer_1;

task body Producer_2 is
begin
     ... -- get data
     Consumer.Process (Data);
     ...
end Producer_2;

task body Consumer is
begin
     ...
     accept Process (D : in Data_Type) do
          ...
     end Process;
     ...
end Consumer;
```

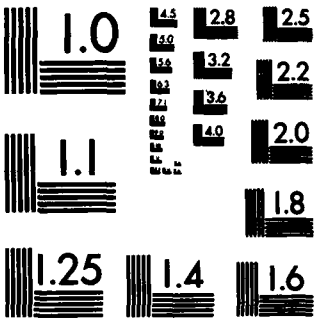
Note that the busy waiting is eliminated, but that now the producers must know which task consumes the data. Note also that we cannot propagate exceptions from producer to consumer should the production of data be hindered, and if the producers terminate the consumer will keep waiting for a call! Thus in this example reversing the direction of entry calls provides us with a tradeoff of a busy wait versus having each of the producers know its consumer.

The decision of the direction of entry call can sometimes be made easier if the coupling between tasks is reduced. If, for example, a buffer were used in the above example, then the consumer and producer tasks need only know about the existence of the buffer. (Presumably the producer will call the buffering task to place data in the buffer and the consumer will call it to get data.) The buffer also minimizes blocking that occurs when a producer is blocked at an entry call until the consumer is ready to rendezvous. While the solution using a buffer seems to be the best, remember that using a buffer will increase both the size and the execution time of the resulting system.

AD-A146 258 REAL-TIME ADA (TRADEMARK)(U) SOFTECH INC WALTHAM MA 3/4
JUL 84 DAAB07-83-C-K514

UNCLASSIFIED F/G 9/2 NL

Cont



Problem

Consider the solution to the Problems in Section 5.1. Notice that the system may be slow to get started because the Scheduler may begin to accept calls to Startup before any of the tasks being scheduled begin execution. Is there a way to fix this?

Solution and Discussion

One solution is to give each of the tasks to be scheduled a higher priority than the neduler itself. Having a higher priority and assuming a fair scheduling algorithm, each of the tasks will reach the call to Scheduler.Startup before the Scheduler starts execution. This has a beneficial side-effect of giving higher priority to the scheduled tasks over the scheduling task all of the time.

An alternate approach is to ensure that during system startup, the Scheduler task is activated after the other tasks, or by adding an entry to the Scheduler task that tells it when to start.

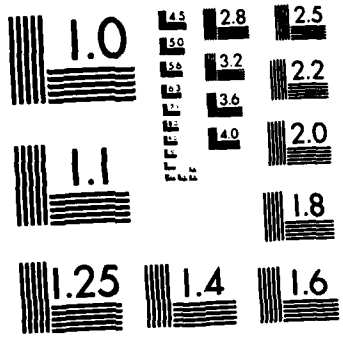
INDEX


EN
ILME

AD-A146 258 REAL-TIME ADA (TRADEMARK)(U) SOFTECH INC WALTHAM MA
JUL 84 DAAB07-83-C-K514

4/4

UNCLASSIFIED F/G 9/2 NL



SUPPLEMENTARY

INFORMATION

DEPARTMENT OF THE ARMY
HEADQUARTERS US ARMY COMMUNICATIONS-ELECTRONICS COMMAND
AND FORT MONMOUTH
FORT MONMOUTH, NEW JERSEY 07703

REPLY TO
ATTENTION OF:

15 OCT 1984

Center for Tactical Computer Systems

Ms. Madeline Crumbacker
Defense Tactical Information Center
Cameron Station
Alexandria, Virginia 22314

Dear Ms. Crumbacker:

As per phone conversation with Ms. Andrea Cappellini, CENTACS on 11 October 1984, a copyright statement has been omitted on documents sent to DTIC and NTIS. Enclosed please find the copyright statement (Encl 1) that must appear in the enclosed list of document (Encl 2). If you have any questions, please contact Ms. Cappellini at 201-544-4280.

Sincerely,

for JAMES E. SCHELL
Director, CENTACS

END

FILMED

12-84

DTIC